

版权注意事项：1、书籍版权归著者和出版社所有；  
2、本PDF仅用于个人获取知识，进行私底下知识交流；  
3、PDF获得者不得在互联网以任何目的进行传播；  
如有需要，请尽量购买正版实体书！支持书籍作者！！



# 运维前线

## 一线运维专家的 运维方法、技巧与实践

云技术社区 著

---

DevOps Fire  
Best Practices From Senior Operation Engineers

---

- 豪华作者阵容，14位来自腾讯、京东、YY、盛大游戏、UC、西山居、猎豹移动等国内知名企业的资深运维专家首次联合，分享他们多年的运维最佳实践
- 精选运维要点，涵盖自动化运维、系统运维、云与虚拟化、Web运维、游戏运维、数据库运维6大主题14个知识点，包含大量企业生产案例



机械工业出版社  
China Machine Press

## 内容简介

本书是运维领域的“集大咖”之作，由运维圈的明星人物、拥有15年运维经验的肖力领衔策划，首次将国内来自腾讯、京东、YY、盛大游戏、UC、西山居、猎豹移动等国内知名企业的14位资深运维大咖聚集在了一起，有针对性地挖掘了他们多年来在各种工作中积累的独到经验和最佳实践。得到了七牛云创始人许世伟、盛大游戏G云总负责人陈桂新等多位业界专家的好评和推荐。

本书是运维领域的“集大成”之作，精选了运维领域重要的6大主题：

- (1) 自动化运维
- (2) 系统运维
- (3) 云与虚拟化
- (4) Web运维
- (5) 游戏运维
- (6) 数据库运维

一共涵盖14个重要的知识点。所有的知识点既有理论的指导，又有方法论的提炼；既有来自这些专家们共事过的企业的商业案例，又有针对企业常见问题的解决方案。

## 云技术社区

成立于2014年，国内最大的云技术交流平台，分享在云计算/虚拟化项目实施中的资讯、经验和技術，坚持干货。旗下运营：云技术实践、云技术、桌面云之云潮涌动等公众号，以及相关的微信群和QQ群。覆盖超过3万人的云计算领域的技术人群。

前线系列

# 运维前线

## 一线运维专家的 运维方法、技巧与实践

云技术社区 著

---

DevOps Fire

Best Practices From Senior Operation Engineers

---



机械工业出版社  
China Machine Press

## 图书在版编目(CIP)数据

运维前线：一线运维专家的运维方法、技巧与实践 / 云技术社区著. —北京：机械工业出版社，2017.1（2017.4重印）

（前线系列）

ISBN 978-7-111-55697-8

I. 运… II. 云… III. ①软件维护 ②数据库系统 ③Linux 操作系统 IV. ①TP311.53  
②TP311.13 ③TP316.85

中国版本图书馆CIP数据核字（2016）第325277号

## 运维前线：一线运维专家的运维方法、技巧与实践

出版发行：机械工业出版社（北京市西城区百万庄大街22号 邮政编码：100037）

责任编辑：何欣阳

责任校对：殷虹

印刷：北京市荣盛彩色印刷有限公司

版次：2017年4月第1版第2次印刷

开本：186mm×240mm 1/16

印张：24.5

书号：ISBN 978-7-111-55697-8

定价：79.00元

凡购本书，如有缺页、倒页、脱页，由本社发行部调换

客服热线：（010）88379426 88361066

投稿热线：（010）88379604

购书热线：（010）68326294 88379649 68995259

读者信箱：hzit@hzbook.com

版权所有·侵权必究

封底无防伪标均为盗版

本书法律顾问：北京大成律师事务所 韩光/邹晓东

## Foreword 推荐序

许式伟，七牛云 CEO。曾就职于金山、百度、盛大，拥有超过 15 年的技术积累，连续 8 年组织 ECUG 实效云计算开发组年会。曾获国家科学技术进步奖二等奖。

2011 年，与吕桂华一起创立七牛云，现已服务 50 多万家企业。

非常荣幸能受邀为本书作序。

2011 年我从盛大离职，创办了七牛云。

在移动互联网时代，初创企业和巨头们之间的力量相差悬殊，这滋生了创业扶持产业的出现与兴起。所谓创业扶持产业，从技术革新上看就是“云”，云服务让所有的初创企业有了和巨头们一样的基础设施。

很快，云服务的时代就在预期中到来了。很多人开始担心，自己的业务知识能否胜任“新运维”的需求。其实我想告诉大家，互联网中各类技术的革新总是很快的，无论你做哪一类工作，当跟不上技术发展的时候，就会面临被淘汰的危险。传统运维的革新是一个必然的过程，如果你对运维知识的了解一直停留在原地，那你离悬崖就不远了。

云服务下的运维相较于传统服务器的模式，优势已相当明显。企业无需花费巨额的资金来购买新的服务器和托管机房的机位，就能够低成本、低风险地实现新增业务。同时，运维人员不再只能利用传统的网管手段来定位系统故障，他们可以通过云计算管理平台以及虚拟设备管理平台进行分析。

此外，云服务的蓬勃发展也令互联网产业发生了巨变，互联网产业整体的蛋糕做大了，这使得其对运维的总体需求呈现上升的趋势。这对运维人员是一个非常好的机会，只要知识与能力满足了岗位需求，你的待遇和发展就能上一个台阶。

作为云服务行业以及创业公司的代表，七牛云深谙创业的艰辛。我们反复说，七牛云的目标是打造一个场景化的 PaaS (Platform-as-a-Service, 平台即服务) 平台，帮助开发者缩短



从想法到产品的距离。我们打造了很多子产品，包括大数据、通用计算、云计算等。从我们发布产品的服务类型来看，第一类是对象存储服务，第二类是融合 CDN 服务，第三类是数据处理服务，第四类是直播云以及实时流网络 LiveNet 服务。

《运维前线》这本书集合 14 位资深运维专家的实践经验，覆盖了互联网和传统行业运维的各个领域。其中所述的运维方法、技巧与实践，都和七牛云息息相关，这也正是我为本书作序的原因。

这里我举两个例子：

在本书第 3 章中讲到，动静分离的架构是基于 Web 开发的互联网服务中常见的架构的，它是指将数据库中的动态内容存储和文件存储分开，常见的做法是将动态内容存储在原有的数据库系统中，而将静态文件，如图片和音视频等，托管存储在七牛云提供的对象存储服务中，这样可以更方便地维护不同类型的数据。

在本书第 12 章中讲到，CDN 节点由于数量众多，承受的流量巨大，再加上国内网络的复杂度极高，因此真正商用 CDN 的建设有一定难度，并且硬件和运维成本都不低，因此一般而言，企业不会去自建 CDN，而是选择七牛云这样的企业来解决问题。

云服务的兴起改变了运维，而这种改变不会停止。

云服务厂商可以将所有事情标准化，然后以服务的形式打包提供给客户。而运维人员将告别烦琐的工作内容，但是他们可能要承担更多的职责——解决监控、评估、采购、报修等问题。

当然，你如果在云服务厂商做运维工作，便需要对传统运维有更深刻的理解了。

## 为什么要写这本书

《运维前线：一线运维专家的运维方法、技巧与实践》(以下简称《运维前线》)是前线系列的一个子集，前线系列图书的出版理念是邀请多位业界专家，总结所在行业的最新理念或深度实践经验。前线系列图书不同于市面上的很多图书，这类书并不系统，有的只是一线专家的实战经验，人们常称之为“干货”。一篇文章、一家公司、一个案例、一个场景，独立成篇，在满足碎片化阅读的同时，也能让读者进行横向比较和深入思考。本系列图书不强调大而全，追求的是每篇文章都是精品，希望能给读者带来深度的启发和收获。

按照这个理念，之前著名产品经理兰军（笔名 Blues）策划的《产品前线》，出版之后大获成功，随之而来的《运维前线》令我感觉到压力巨大。《运维前线》的出版犹如十月怀胎，中间充满波折，好几次我都以为要半途而废了，最后终于得以出版，在此要特别感谢机械工业出版社华章公司著名出版人杨福川，是他的坚持和鼓励让我总能在迷茫中看到希望。我和福川有共同的理念，希望把《运维前线》做成精品，如果有哪点不符合要求，那么我们宁愿耗时长一些，多打磨打磨，很庆幸能与福川一起合作。

本书共有 14 位作者，包含了在腾讯、YY 语音、UC、京东、盛大游戏、金山西山居、猎豹移动、广发银行、优维科技等多家公司工作的实践经验，基本覆盖了互联网和传统行业运维各个领域，估计这是迄今为止第一本由这么多资深运维专家联合写成的图书，也是第一本分享了众多一线运维专家亲身实践的图书。本书的出版也充分体现了互联网开放合作的精神。

看到本书的目录时，我激动不已，即使书中的内容我已经看了好多遍，但是在回顾目录的时候，我依然感到这是一本非常有吸引力的书，是一本每名运维工程师都应该案头常备的书！

## 本书特色

当前 IT 领域的概念层出不穷，云计算、物联网、移动互联网、大数据、人工智能、VR，所有的这一切都基于 IT 系统，IT 系统正在向规模更大、更复杂、更高级的方向演进，一切 IT 资源都掌握在运维手里，通过运维来操作。这个时代对运维的要求越来越高，运维如果稍有不慎，就会造成重大的损失，所以随着 IT 系统的发展，运维的重要性也越来越高。

根据量子力学理论，世界由基本粒子组成，因此世界是不连续的，这个理论在运维知识体系的建立上同样适用。仔细回想一下自己运维体系的建立，就是逐个攻克和掌握知识点，再进一步通过实践不断加深的过程。《运维前线》也是这样，其中的每一篇文章都能够协助读者更快地掌握一个或多个知识点，相信通过运维前线系列的逐步出版，最终能够覆盖更多的运维知识点，为读者带来更多的实践经验和理念。

作为运维前线系列的第一本书，本书覆盖了运维自动化、系统运维、云及虚拟化、Web 运维、游戏运维、DBA 运维等 6 个方面 14 个知识点，都是各位作者总结自己多年实践经验的干货，每一篇文章都很精彩，都值得读者仔细品味。

## 本书作者

本书第 1 章“自动化运维之深度解码”，来自订阅号“互联网运维杂谈”的作者、优维科技创始人王津银，人称“老王”。我在多个场合听过“老王”的分享，有 40 分钟的，也有长达 3 个小时的，令我惊讶的是，“老王”每次总能带来新的东西。这篇“自动化运维之深度解码”也是如此，凝结了老王许多最新的见解，值得深读。

胥峰是我在盛大游戏的前同事，从胥峰身上我学到了不少知识，比如解决问题的思路和方法，有时候碰到运维难题，也许换一个角度就能迎刃而解。

刘宇、尹会生、陈立军是我多年的同事。刘宇、会生已经出版了多本运维图书，他们都是非常资深的运维专家。刘宇无论演讲还是文章都逻辑清晰，丝丝入扣。会生和立军分享的都是我亲眼所见的、在工作中碰到的难题及解决方法。

张观石是欢聚时代（YY）互娱事业部业务运维负责人，有多年的将 PHP 运用到日常运维中的经验。观石将 PHP 用到了极致，即使不懂 PHP，也可以通过本书一窥观石在运维方面的丰富经验。

马亮有多年的游戏运维经验，目前在腾讯云专注做游戏云的架构设计，对游戏运维有深刻的理解。

本书的作者还有冉宏元（老男孩）、余洪春（抚琴煮酒）、吴传玉、彭华盛、蒋迪、赵旻、赵海军。虽然我与他们未曾谋面，但是彼此都是熟悉的网友，他们的文章我都曾仔细拜读并



多次请教过，其中的运维思想让我深深折服，非常期待能有机会向他们当面请教。

## 读者对象

本书面向所有的运维工程师，无论是资深运维，还是刚入行的运维，相信都能从本书中获益。本书的读者对象包括如下几类：

- ☐ 系统运维工程师
- ☐ 安全工程师
- ☐ 数据库运维工程师
- ☐ 业务运维工程师
- ☐ 网络运维工程师
- ☐ 运维系统开发工程师及架构师
- ☐ 云计算 / 虚拟化运维工程师
- ☐ 其他对运维感兴趣的读者

## 勘误和支持

由于作者的水平有限，编写时间比较仓促，书中难免会出现一些错误或不准确的地方，恳请读者批评指正。如果您有更多宝贵的意见，欢迎关注我的订阅号“云技术实践”，然后在后台将您的宝贵意见发送给我。本书的勘误也会通过订阅号进行发布，关注订阅号请扫描下面的二维码：



“运维前线”是一个系列，欢迎在平时工作中积累了实战经验的广大运维工程师继续参与“运维前线”的写作，带来更多的干货分享！

期待能够得到你们的真挚反馈，在运维之路上互勉共进。

## 致谢

从开始构思到《运维前线》的出版，本书的创作过程犹如十月怀胎，期间也获得了无数的支持与肯定。借此机会，向所有为此奉献力量的人表示深深的感谢。

感谢参与本书写作的 14 位行业专家，他们在百忙之中伏案写作，将自己的经验共享给

广大读者。能够和你们合作，我深感荣幸，经过将近一年时间的沟通和交流，你们的专业和执着深深地打动了我，同时也让我学到了不少东西。

感谢机械工业出版社华章公司的首席策划杨福川，编辑高婧雅、孙海亮。在近一年的时间中，你们的敬业精神不断地鼓舞着我前进，你们的鼓励、帮助和坚持引导了本书的完成。

这里，我还要特别感谢七牛云。七牛云是我见过的最具技术气质的云服务企业，肩负“帮助开发者缩短从想法到产品的距离”的使命，立志打造以数据为核心的场景化 PaaS 服务。七牛云主张技术共享并热衷于技术的传播，发起架构师实践日，推出各类创业扶持计划，这些都是七牛云正在做的事。

## 特别致谢

本书在成书过程中，得到了一批热心志愿者的协助，他们是陈家豪、曹学朋、邓荣兴、刘海文、李斯朗、韩海林，尤其刘海文做了大量的义务工作，在此特别感谢！

肖 力

## 志愿者的话

能在力哥组织的这本书中担任助理，我感到非常荣幸。本书出自一群经验老到并乐于分享的运维前辈之手，是一本案头必备的好书。

刘海文

## Contents 目 录

推荐序

前 言

### 第 1 章 自动化运维之深度解码 ..... 1

1.1 概述 ..... 1

1.2 运维自动化的三重境界 ..... 3

1.3 运维自动化的困境和价值 ..... 4

1.3.1 运维自动化的困境 ..... 4

1.3.2 运维自动化的价值 ..... 4

1.4 运维自动化的多维解读 ..... 5

1.4.1 基于应用变更场景的维度划分 ..... 5

1.4.2 基于系统层次的维度划分 ..... 8

1.4.3 基于与业务程序耦合紧密程度的  
维度划分 ..... 8

1.4.4 面向服务的自动化能力划分 ..... 9

1.5 运维自动化的方法论 ..... 11

1.6 运维自动化系统的实现 ..... 12

1.6.1 DNS 管理系统 ..... 12

1.6.2 CMDB 管理系统 ..... 13

1.6.3 名字服务中心系统 ..... 13

1.6.4 持续部署管理系统 ..... 14

1.6.5 运维调度管理系统 ..... 15

1.7 运维自动化系统的 API

参考实现 ..... 16

1.8 运维自动化依赖的团队模型 ..... 17

1.8.1 团队的能力模型 ..... 17

1.8.2 团队的驱动模型 ..... 18

1.8.3 团队的技能模型 ..... 18

1.8.4 参考的运维团队组织结构 ..... 19

1.9 小结 ..... 19

### 第 2 章 利用 Factor 和 Django 快速 构建 CMDB ..... 20

2.1 CMDB 简介 ..... 21

2.2 开源 CMDB 介绍 ..... 21

2.2.1 OneCMDB 介绍 ..... 21

2.2.2 CMDBuild 介绍 ..... 22

2.2.3 其他的开源 CMDB ..... 23

2.3 Puppet 及 Factor 介绍 ..... 24

2.3.1 什么是 Puppet ..... 24

2.3.2 为什么是 Puppet ..... 25

2.3.3 什么是 Factor ..... 25

2.3.4 Factor 的特点 ..... 25

2.3.5 Factor 变量 ..... 26

2.4 如何利用 Python 获取 Facts .....	27	3.2.5 批量更改虚拟机名称及 port group .....	69
2.4.1 工作原理 .....	27	3.2.6 批量设置虚拟机版本和 CPU、 内存保留值 .....	70
2.4.2 利用 Python 脚本获取 Facts .....	27	3.3 利用批处理与 Shell 脚本简化逻辑 节点的搬迁 .....	71
2.5 使用 Django 快速构建 CMDB 系统 .....	28	3.3.1 逻辑节点切换脚本的思路 .....	71
2.5.1 Django 介绍 .....	28	3.3.2 利用批处理脚本简化 Windows 逻辑节点的搬迁 .....	72
2.5.2 Django 安装 .....	29	3.3.3 利用 Shell 脚本简化 Linux 逻辑 节点的搬迁 .....	78
2.5.3 Django 常用命令 .....	30	3.3.4 通过 SFTP 和 WMIC 指令将脚本 文件上传至所有虚拟机 .....	86
2.5.4 Django 的配置 .....	30	3.3.5 搬迁期间的注意事项 .....	87
2.6 高级进阶 .....	44	3.4 小结 .....	87
2.6.1 历史查询功能 .....	44		
2.6.2 API 功能 .....	45		
2.6.3 数据表结构 .....	48		
2.6.4 用户管理功能 .....	50		
2.6.5 用户组管理功能 .....	51		
2.7 小结 .....	53		
<b>第 3 章 数据中心搬迁中的 x86 自动化运维 .....</b>	<b>54</b>	<b>第 4 章 集中配置管理工具 Puppet .....</b>	<b>88</b>
3.1 数据中心搬迁准备 .....	54	4.1 如何同步 puppet-agent 端上的常用 服务 .....	89
3.1.1 数据中心搬迁介绍 .....	54	4.2 如何在 puppet-agent 端上自动 安装常用的软件包 .....	90
3.1.2 搬迁环境介绍 .....	55	4.3 如何自动同步 puppet-agent 端的 yum 源 .....	90
3.1.3 搬迁前的准备工作 .....	56	4.4 如何根据不同名字的节点机器推送 不同的文件 .....	92
3.1.4 搬迁信息收集 .....	56	4.5 如何根据节点机器名来选择性地 执行 Shell 程序 .....	95
3.2 利用 VMware 脚本简化虚拟化层的 搬迁 .....	58	4.6 如何快速同步 puppet-server 端的 www 目录文件 .....	97
3.2.1 通过脚本完成 ESXI 安装后的 基础设置 .....	58	4.7 如何利用 ERB 模板自动配置 Apache 虚拟主机 .....	102
3.2.2 批量挂载数据盘 .....	63		
3.2.3 批量注册虚拟机 .....	67		
3.2.4 vCenter 目录结构的调整 .....	68		

4.8 如何利用 ERB 模板自动配置 Nginx 虚拟主机 .....	105
4.9 小结 .....	110

## 第 5 章 深度实践 iptables .....

5.1 禁用连接追踪 .....	111
5.1.1 排查连接追踪导致的故障 .....	111
5.1.2 分析连接追踪的原理 .....	113
5.1.3 禁用连接追踪的方法 .....	114
5.1.4 确认禁用连接追踪的效果 .....	117
5.2 慎重禁用 ICMP 协议 .....	117
5.2.1 禁用 ICMP 协议导致的一则 故障案例 .....	117
5.2.2 MTU 发现的原理 .....	119
5.2.3 解决问题的方法 .....	121
5.3 网络地址转换在实践中的案例 .....	121
5.3.1 源地址 NAT .....	121
5.3.2 目的地址 NAT .....	122
5.4 深入理解 iptables 的各种表和 各种链 .....	123
5.5 小结 .....	125

## 第 6 章 使用 systemd 管理 Linux 系统服务 .....

6.1 systemd 和 sysVinit 之间的关系 .....	126
6.1.1 sysVinit 方式下系统的启动 特点 .....	127
6.1.2 systemd 方式下系统的启动 特点 .....	127
6.2 systemd 的原理和启动顺序 .....	128
6.2.1 sysVinit 的启动顺序 .....	128

6.2.2 systemd 的启动顺序 .....	130
6.3 systemd 的进程控制命令 .....	135
6.3.1 systemctl 命令 .....	136
6.3.2 hostnamectl 命令 .....	136
6.3.3 localectl 命令 .....	137
6.3.4 loginctl 命令 .....	137
6.3.5 timedatectl 命令 .....	138
6.4 systemd 服务管理 .....	138
6.4.1 编写 Nginx 的 sysVinit 启动 脚本 .....	138
6.4.2 编写 Nginx 的 systemd 启动 脚本 .....	140
6.4.3 systemd 的其他功能 .....	142
6.5 优化 .....	146
6.5.1 使用 systemd-analyze 优化 启动时间 .....	146
6.5.2 使用 systemd journal 功能 .....	148
6.6 小结 .....	148

## 第 7 章 PHP 运维实践 .....

7.1 PHP 再认识 .....	150
7.1.1 PHP 进程的工作方式 .....	150
7.1.2 PHP 代码的编译和部署 .....	151
7.1.3 PHP 内部实现和生命周期 .....	151
7.1.4 PHP 在互联网技术栈的 位置 .....	152
7.2 PHP 开发、架构、运维问题及 解决思路 .....	153
7.2.1 运维对 PHP 研发提要求 .....	153
7.2.2 运维参与 PHP 项目架构 设计 .....	154

7.2.3	PHP 运维常见问题及解决之道	156
7.3	PHP 进程部署和配置、代码发布	157
7.3.1	PHP 进程的部署	157
7.3.2	PHP 配置文件变更	161
7.3.3	PHP 配置项	162
7.3.4	PHP 进程部署及配置文件管理实践	164
7.3.5	PHP 代码发布	165
7.3.6	PHP 代码发布实践：代码发布系统	167
7.4	PHP 性能分析	170
7.4.1	性能问题概述	170
7.4.2	PHP 性能问题	171
7.4.3	性能分析方法	172
7.4.4	PHP 性能分析实践：性能分析系统	181
7.5	PHP 故障处理与监控	182
7.5.1	PHP 故障分类及处理思路	183
7.5.2	业务监控和故障发现	184
7.5.3	PHP 故障消除的方法	186
7.5.4	故障分析案例	187
7.6	小结	189

## 第 8 章 应用系统运行分析 190

8.1	分析模型	191
8.1.1	数据采集	191
8.1.2	数据模型	194
8.2	运行分析平台建设	199

8.2.1	数据采集接口	199
8.2.2	数据分析模块	200
8.2.3	推广	200
8.3	呼叫中心系统运行分析示例	201
8.3.1	确定分析方案	201
8.3.2	问题分析案例介绍	202
8.4	小结	203

## 第 9 章 虚拟化中存储配置典型

### 场景：启动风暴 204

9.1	oVirt 虚拟化平台配置介绍	205
9.1.1	存储配置背景知识	205
9.1.2	模板与实例同一存储	206
9.1.3	模板与实例分离存储	207
9.1.4	无状态实例的硬盘与快照分离存储	207
9.2	启动风暴相关系列实验	208
9.2.1	模板配置	208
9.2.2	实验脚本	208
9.2.3	WD 1TB 机械硬盘启动 Windows XP 实验	210
9.2.4	Intel 480GB SSD 启动 Windows XP 实验	212
9.2.5	实验结论	214
9.3	私有云中处理启动风暴的常用方法	214
9.3.1	启动排队	214
9.3.2	存储分层选择	215
9.3.3	其他提升桌面云存储性能的方式	217
9.4	小结	219

## 第 10 章 私有云桌面网络组建 ····· 220

### 10.1 桌面云常用网络 ····· 220

#### 10.1.1 NAT 网络 ····· 220

#### 10.1.2 桥接网络 ····· 223

#### 10.1.3 VLAN 网络 ····· 226

#### 10.1.4 Open vSwitch ····· 231

### 10.2 oVirt/OpenStack 的桌面网络应用 ····· 232

#### 10.2.1 oVirt/OpenStack 组网方式 ····· 232

#### 10.2.2 应用场景举例 ····· 237

### 10.3 小结 ····· 239

## 第 11 章 浅谈服务器交付的那些事儿 ····· 240

### 11.1 设备签收的学问 ····· 240

### 11.2 服务器设置 ····· 241

### 11.3 Cobbler 的流程与规划 ····· 244

### 11.4 服务器安装时遇到的各种坑 ····· 247

#### 11.4.1 DHCP 客户端获取 IP 地址失败 ····· 247

#### 11.4.2 TFTP 加载失败 ····· 248

#### 11.4.3 TFTP Client 交互后无响应 ····· 248

#### 11.4.4 yum 安装失败 ····· 249

#### 11.4.5 Linux 内核无法识别新硬件 ····· 250

#### 11.4.6 恶意 PXE 启动导致原有系统被误装 ····· 250

### 11.5 交接后的故事 ····· 250

### 11.6 小结 ····· 252

## 第 12 章 企业级 Nginx Web 服务优化实战 ····· 254

### 12.1 Nginx 基本安全优化 ····· 254

#### 12.1.1 调整参数隐藏 Nginx 软件版本号信息 ····· 254

#### 12.1.2 更改源码隐藏 Nginx 软件名及版本号 ····· 256

#### 12.1.3 更改 Nginx 服务的默认用户 ····· 259

### 12.2 根据参数优化 Nginx 服务性能 ····· 260

#### 12.2.1 优化 Nginx 服务的 worker 进程个数 ····· 260

#### 12.2.2 优化绑定不同的 Nginx 进程到不同的 CPU 上 ····· 262

#### 12.2.3 Nginx 事件处理模型优化 ····· 265

#### 12.2.4 调整 Nginx 单个进程允许的客户端最大连接数 ····· 266

#### 12.2.5 配置 Nginx worker 进程的最大打开文件数 ····· 267

#### 12.2.6 优化服务器域名的散列表大小 ····· 267

#### 12.2.7 开启高效文件传输模式 ····· 269

#### 12.2.8 优化 Nginx 连接参数, 调整连接超时时间 ····· 269

#### 12.2.9 上传文件大小的限制 (动态应用) ····· 272

#### 12.2.10 FastCGI 相关参数调优 (配合 PHP 引擎动态服务) ····· 273

12.2.11	配置 Nginx gzip 压缩 实现性能优化 .....	277	12.10.1	什么是 CDN .....	302
12.2.12	配置 Nginx expires 缓存实现 性能优化 .....	279	12.10.2	CDN 的特点 .....	303
12.3	Nginx 日志相关的优化与安全 .....	283	12.10.3	企业使用 CDN 的基本 要求 .....	304
12.3.1	编写脚本实现 Nginx access 日志轮询 .....	283	12.11	Nginx 程序架构优化 .....	304
12.3.2	不记录不需要的访问日志 .....	284	12.12	使用普通用户启动 Nginx (监牢模式) .....	305
12.3.3	访问日志的权限设置 .....	284	12.12.1	为什么要让 Nginx 服务 使用普通用户 .....	305
12.4	Nginx 站点目录及文件 URL 访问控制 .....	284	12.12.2	给 Nginx 服务降权的 解决方案 .....	305
12.4.1	根据扩展名限制程序和文件 访问 .....	284	12.12.3	给 Nginx 服务降权 实战 .....	306
12.4.2	禁止访问指定目录下的所有 文件和目录 .....	285	12.13	控制 Nginx 并发连接数量 .....	308
12.4.3	限制网站来源 IP 访问 .....	286	12.14	控制客户端请求 Nginx 的 速率 .....	312
12.4.4	配置 Nginx, 禁止非法域名 解析访问企业网站 .....	287	12.15	小结 .....	314
12.5	Nginx 图片及目录防盗链解决 方案 .....	288	<b>第 13 章 游戏运维的思考 .....</b>	<b>315</b>	
12.6	Nginx 错误页面的优雅显示 .....	295	13.1	游戏运维最关键的几件事 .....	315
12.6.1	生产环境中常见的 HTTP 状态码列表 .....	295	13.1.1	安全 .....	315
12.6.2	为什么要配置错误页面优雅 显示 .....	295	13.1.2	稳定 .....	318
12.7	Nginx 站点目录文件及目录权 限优化 .....	298	13.1.3	高效 .....	322
12.8	Nginx 防爬虫优化 .....	300	13.1.4	成本节约 .....	323
12.9	利用 Nginx 限制 HTTP 的 请求方法 .....	302	13.2	游戏运维人的发展 .....	325
12.10	使用 CDN 做网站内容加速 .....	302	13.3	小结 .....	326
			<b>第 14 章 数据库平台建设实战 .....</b>	<b>327</b>	
			14.1	规范建立 .....	327
			14.1.1	安装规范 .....	328
			14.1.2	配置规范 .....	329



14.1.3 账号、权限规范 .....	335	14.3.2 日志部分 .....	349
14.1.4 目录规范 .....	336	14.3.3 资产部分 .....	351
14.1.5 其他规范 .....	337	14.3.4 信息展示 .....	353
14.2 架构设计 .....	339	14.3.5 入口 (LVS/Redir) .....	354
14.2.1 架构图 .....	339	14.4 后期功能展望 .....	357
14.2.2 各个模块介绍 .....	340	14.5 小结 .....	357
14.3 功能介绍与实践 .....	341		
14.3.1 操作部分 .....	341	附录 A 求职者与面试官 .....	358

# 自动化运维之深度解码

## 作者简介

王津银，2005 年硕士毕业，参与电信 BOSS 系统研发两年。而后于 2007 年进入腾讯公司接触运维，经历服务器从百到万的运维历程，先后在 YY 和 UC 参与不同业务形态的运维，期间带过前端运维、数据存储运维、YY 语音、游戏运维、运维研发等多种运维团队，对运维有着全面的理解。极力倡导互联网价值的运维理念，即面向用户的价值是由自动化平台来交付和传递，同时由数据化来提炼和衡量的。“精益运维”理论的创始人。个人微信公众号“互联网运维杂谈”(waynewang\_ops)，粉丝 2.5 万人，现创办优维科技公司，旨在缩短企业到达互联网运维的路径。

自动化运维是一个人让人兴奋且容易失控的话题！兴奋是因为我想做一次尝试，把它的全貌和细节说清楚；容易失控是因为涉及点太多，一则怕遗漏，二则怕顾此失彼。带着这份复杂的心情，我们来一次自动化运维的解析之旅吧。说实话，一个运维团队的运维能力究竟如何，其实看一个自动化管理系统便可得知！

## 1.1 概述

作为开篇，首先让我们来熟悉一下运维全平台的规划体系吧，如图 1-1 所示。

很多人看到这样一个架构图，可能会纳闷，难道对于一个小型企业来说，也要实施如此复杂的运维自动化体系吗？其实，对于不同规模的企业来说，对运维自动化的诉求的确是不同的。对于大规模企业，如 BAT，这些能力基本上都是必不可少的；而对于小型互联网企业，比如



图 1-1 运维全平台规划体系

说 App 开发公司，则核心的自动化诉求可能更多的是配置管理工具，比如说 Puppet、SaltStack 或 Jenkins+Rsync 等。

我们不禁要问，有什么样的准则可以让我们作为依据来判断何时该如何导入自动化？应该导入自动化的哪些部分？当你需要持续、频繁地进行一些事情时，此时就需要引入自动化，比如说版本发布，如果这个时候你感觉到很痛苦，那么此时就需要引入自动化了。关于应该导入自动化的哪些部分，我个人的经验是根据角色去梳理他的工作现状（持续、频繁的工作），然后引入自动化的能力，再根据角色人数的多与少来确定事情的优先级，比如说系统管理和业务发布，很明显业务发布的优先级更高，因为它的自动化所带来的人力解放的收益更大。当然还有一种更理想的情况，那就是根据整体业务交付流来构建，以它的全流程自动化为目标，此时引入的是该交付链上所有的自动化能力，当然对于很多企业来说，这种自动化实现的代价很高，而得到的收益却很小。

## 1.2 运维自动化的三重境界

宋代禅宗大师青原行思（六祖门下首座）提出参禅的三重境界：

参禅之初，看山是山，看水是水；

禅有悟时，看山不是山，看水不是水；

禅中彻悟，看山仍然山，看水仍然是水。

这三重境界其实和我们眼中运维自动化的三重境界是类似的。

**运维自动化第一重境界：看山是山，看水是水。**开始接触运维自动化的时候，我们看到了很多工具认为它们就代表着自动化，比如说早期将 Expect+SSH 封装在一起之后，就认为可以实现批量运维了。看到有人说 Puppet 可以做配置管理，这个时候就会认为 Puppet 可以做配置管理，甚至是发布管理。这个时期的典型问题就是以偏概全，对于某个开源自动化工具来说，还没法去界定它的使用场景和范围，这样将直接影响系统的建设效益。这个时候才开始知道我们看到的山不是真正的山，而是迷雾环绕的深山。

**运维自动化第二重境界：看山不是山，看水不是水。**此时我们已经知道只有 Expect+SSH 还不够，随着业务规模的变化，我们还需要一个更完整的概念来做发布系统，真正的发布系统要做版本管理、环境管理、配置管理，还有生命周期管理等；配置管理工具想让自动化变得更加完美，其实还要依赖于 OS 和应用层的标准化规范，比如说应用交付规范、应用打包规范、OS 的统一等等。对于其他资源对象的管理来说，生命周期的概念均穿行其中，比如说 DNS、LVS、接口、配置、应用包等。为了有效地标识资源的生命周期状态，需要用大量的数据来实时反馈。这是运维自动化更具体的层面，将一个个的山貌都看清楚了。

**运维自动化第三重境界：看山还是山，看水还是水。**这是一种自动化本质上的追究，站在高山之巅，俯览众山，会发出原来如此的感叹：所有自动化的本质都是为了可视化，让所

有的人看到一致的服务，从而确保结果一致；从底层来说，你可以认为所有自动化的本质都是指令 + 文件分发的组合；你会进一步抽象系统的运维自动化能力，提供即插即用的机制；结合服务化的需求，进一步云化所有的运维系统，确保内外使用的一致性，最终自动化的平台就是一个整合的持续交付平台。这是化境！

## 1.3 运维自动化的困境和价值

### 1.3.1 运维自动化的困境

谈到运维自动化的困境，我觉得要带着两大行业特点去看待这个问题，一个是互联网行业、另外一个就是传统行业。这两个行业面临的运维自动化的困境完全不同，普遍的共性是运维研发资源能力的不足。

对于互联网行业，业务的发展速度很快，底层运维自动化能力可通过 IaaS 公有云来解决。在 OS 之上的运维自动化，则是通过一些开源工具来解决的，比如说 Puppet、SaltStack、Ansible 等。大部分都是以开源工具为主，开源产品的引入，也在不断加大维护的难度和复杂度，带来的另外一个问题就是平台可扩展的能力非常弱。所以一般成规模的互联网企业，最后都走向了自研的道路。不过有利于互联网行业运维平台建设的条件是互联网的基础比较标准，在硬件和软件的差异上不像传统企业那么大。

对于传统行业，业务的互联网是瞬间展开的，另外传统的封闭式系统架构也走向了开放式 x86 架构，导致运维维护的基础设施对象和上层的业务对象提升了一个数量级。而传统企业的运维手段之前都依赖于商业产品和人肉运维等方式。

无论是成规模的互联网企业还是传统企业，在业务的倒逼之下，运维的突破力都是不断向前的，但是这个整体的规划蓝图是什么样的、实施路径如何、需要什么样的方法论，则需要有一个全面的解答。

### 1.3.2 运维自动化的价值

谈到运维自动化的价值，运维人员应该很容易就能达成我所说的如下共识。

首先是效率的提升、人力的解放，通过工具或平台来提升人均的运维效率和产出，比如说之前通过人肉发布一天只能发布 10 张单，现在通过工具，一天可以发布 100 张单。

其次提升了产品的交付效率，提高了业务的竞争能力。快是制胜的法宝之一，如果你的产品推出得比对方更快，那么你就能更快地接触到用户或客户。

还能提升产品的质量，通过工具不断去提高持续交付链上各角色的能力，比如说测试组的自动化测试、配置管理组的持续集成服务，等等，通过能力的整合，不断提高软件交付的质量；还有在发生故障的时候，能有更快的恢复手段来确保故障的恢复，也是质量保障的一部分。

最后运维自动化的收益是成本的节省，一种是最直接的人力成本的节省，可以让更少的人做更多的事情；间接的成本受益是把很多运维经验固化成平台的经验，从而减少了整个交付链上的文档化内容的输出。

## 1.4 运维自动化的多维解读

### 1.4.1 基于应用变更场景的维度划分

我们曾经探讨过，所有运维的价值导向最终都是面向业务、面向用户，所以自然而然就需要从业务的维度进行划分。而运维是有很多种场景的，但从业务的角度来说，核心的业务场景一般就包括如下5种：业务上线、业务下线、业务扩容、业务缩容和应用升级。下面将以其中一种场景为例，将整个流程穿起来看看，以此识别流程的节点到底对接了哪些系统？针对其他的业务场景，我们也可以用同类的方法进行分析。首先预设业务的架构如图1-2所示。

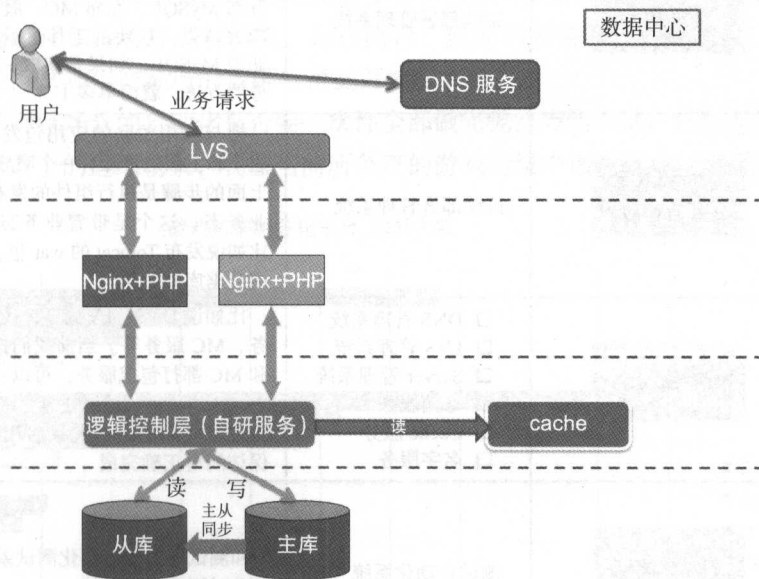


图 1-2 业务架构示例图

(1) **业务上线**。表示上线一个完整的应用。从无到有部署整个业务上线，具体的流程如图1-3所示。

仔细看图1-3中所示的流程，我们会发现该流程涉及多个系统，每个系统所完成的职能又都有不同，这里只是大概地描述了一下。但一旦将这个流程清晰地梳理出来，我们就能知道真正地将一个应用全部上线到底有多复杂了。但看完图1-3又会觉得其实比较简单了，因为从业务上线的流程来看，我们只需要一个上层的流程调度引擎再加上对应的执行器，执行



业务上线流程图			
	阶段	关联系统	完成职责
服务器申请	资源申请	<input type="checkbox"/> CMDB <input type="checkbox"/> 公有云 IaaS 资源 <input type="checkbox"/> 虚拟机平台	从资源池中获取到需要的 OS。对于传统非云模式，此时 CMDB 一般作为资源池。而对于云模式，需要申请的则及时申请，采用按需申请的模式
服务器初始化	OS 环境初始化	配置管理工具，类似于 Puppet、SaktStack 或其他	无论是从资源池获取服务器还是从云端获取资源，此时不同的业务对 OS 环境的依赖也有所不同，比如说一些内核参数和用户等
组件部署	发布组件	持续部署管理系统	根据应用的需要去执行相应的部署，比如说发布 Nginx、发布 Tomcat、发布 MySQL、发布 MC。服务的云端能力越强，这块的工作量就越小，比如说 MySQL，如果是获取一个存储服务的方法，就简单多了
业务环境初始化	发布对应的应用程序包	持续部署管理系统	把与应用关联的应用包发布出去，这个和上面的步骤有一个明显的区别，上面的步骤是执行组件的发布，不带业务态。这个是带着业务态的执行。比如说发布 Tomcat 的 war 包、MySQL 的数据库初始化
关联服务申请	申请关联服务	<input type="checkbox"/> DNS 管理系统 <input type="checkbox"/> LVS 管理系统 <input type="checkbox"/> Server 管理系统 <input type="checkbox"/> 存储服务 <input type="checkbox"/> Cache 服务 <input type="checkbox"/> 名字服务	比如说 DNS、LVS、MySQL 存储服务、MC 服务等。当前我们把 MySQL 和 MC 都打包成服务，可以一键申请；PaaS 系统精髓也就在这儿。名字服务，把业务之间的访问关系启用起来，确保访问可正确完成
自动化测试	自动化测试	测试自动化系统	和测试那边的自动化测试系统对接，自动对测试进行对接
上线	业务上线	<input type="checkbox"/> 监控系统 <input type="checkbox"/> 数据采集系统 <input type="checkbox"/> CMDB 系统	监控系统启用监控、数据采集系统开始采集数据、CMDB 沉淀系统的配置，比如说机器上运行了什么应用，什么进程等

图 1-3 业务上线流程

器通过 API 和底层的各个系统对接即可。这也是为什么之前在框架图(图 1-2)中,要求各个专业系统一定要向上提供 API,并且要求这个 API 的风格必须是一致的。

最复杂的业务上线流程梳理完成之后,业务下线其实很简单,它是上线过程的逆过程,上线负责装,下线负责拆。

业务上线之后,随着用户活跃度的上升,业务的容量逐渐会出现不足的情况,此时就需要进行业务扩容。业务扩容其实很简单,当某类节点出现不足的时候,就对它进行扩容。业务扩容所要做的变更,其实都是业务上线的子流程。比如说如果 Web 层容量不够,那就申请机器,安装组件、下发应用包,进行自动化测试。这个时候需要注意的是:在业务上线的过程中,我们把很多的配置信息都下放到 CMDB 中了,因此我们在选择扩容的时候,就要从 CMDB 中把信息读取出来,以指导变更。

应用升级,目前持续集成所讲的自动化都集中在这块。简单来讲,就是升级程序包、升级配置、执行额外的指令等,一般来说逃脱不了这几种模式。读者可能会问,如果正如你所说的这么简单,那么是不是将 SSH 封装成一个 UI 就可以了。当然不是,这个时候还需要你对运维的理解,在底层进行一些标准化的工作,否则你提供的就只是一个工具,而完全没有运维的思路,比如说程序运行属主、运行路径、监控的策略等。另外建设应用发布平台的目的是要让测试和生产环境的运维变得更可控。

以上几个运维场景的自动化是否要一次性全部做完呢?当然不是,它们也是有先后和主次之分的。对于以上的运维场景,我在当前所负责的游戏运维中做过统计,数据如图 1-4 所示。

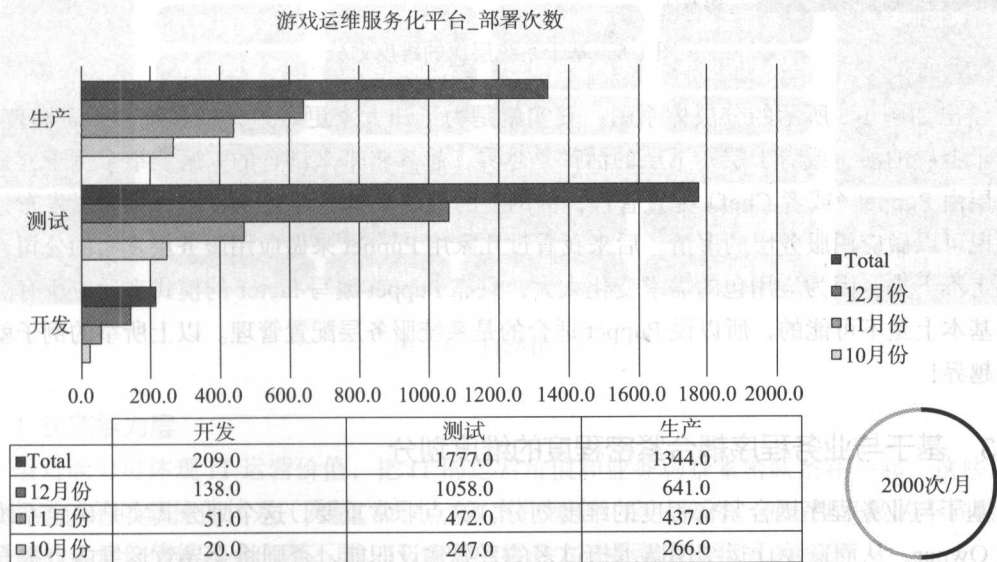


图 1-4 持续部署的数量

(2) 持续部署的数量,一个月 2000 次左右。



(3) 其他场景的分布情况。一个月上线一次、下线两次、扩容一次左右。

有了这个数据，我们在建设一个自动化系统的时候，就能意识到应该先做什么后做什么。当然，不同的企业有不同的实际情况，还是应该找到核心痛点，而不是一上来就建设完整的业务变更系统，那样不仅见效不快，且容易让项目收益不大，从而遇到很大的阻力。

## 1.4.2 基于系统层次的维度划分

首先来看下系统层次的维度划分，如图 1-5 所示。

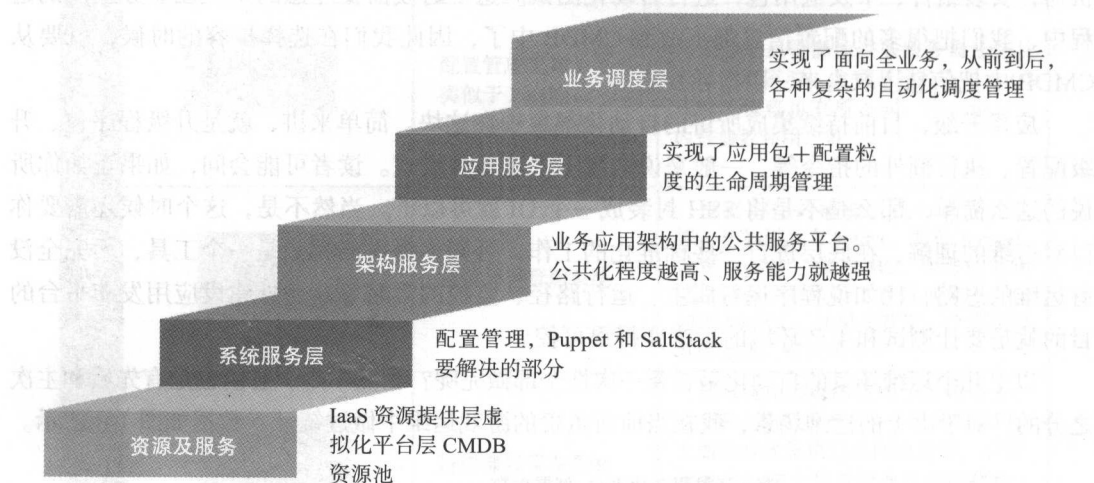


图 1-5 基于系统层次的维度划分

给出如图 1-5 所示的分层体系图，其实就是为了让大家更好地识别系统的职责和范围，下层干上层的活，或者上层干下层的活都是越界，越界将带来耦合的问题。举个例子，系统服务层由 Puppet（或者 Chef）配置管理，但网上的很多资料都说 Puppet 可以用来做发布，也就是说可以做应用服务层的事情。后来我看过几家用 Puppet 来做应用服务层发布的公司，最后都走不下去，因为应用包的需求变化太大，只靠 Puppet 编写 factor 的模式来适应所有的场景，基本上是不可能的，所以说 Puppet 适合的是系统服务层配置管理。以上所举的例子就是一种越界！

## 1.4.3 基于与业务程序耦合紧密程度的维度划分

基于与业务程序耦合紧密程度的维度划分，这点非常重要，这个划分其实是确定系统建设的 Owner，从而避免让运维团队承担过多的系统建设职能，否则将会导致运维能力提升缓慢。那么应该如何判断与业务程序耦合的紧密程度呢？我的准则非常简单，线上程序直接调用的就是紧耦合，或者由研发主导的公共服务，类似于 API/SDK 类的后端服务，应该由测试来主导系统建设；有些服务与程序不是直接关联的，或者是由运维牵头建设的，则由运维来

主导，例如 LVS、DNS 服务等。

有这样一种情况，在很多应用程序中，DNS 和 LVS 服务也存在于程序调用链中，怎么办？在我的方案中，绝对不允许内部服务走 DNS 和 LVS。我们都知道 DNS 和 LVS 的服务对于服务异常的处理（DNS 无状态、LVS 是七层能力弱），远远达不到线上服务的要求，所以要坚决拒绝。如果真的有人要使用 DNS 和 LVS，那么第一告诉他们业务的风险；第二，发生故障的时候，需要让研发参与处理。另外这也是系统的边界没划分清楚的问题，是让运维组件去承担业务上应该具备的容灾容错功能，这会令后面的运维系统建设增加很多不必要的功能。

#### 1.4.4 面向服务的自动化能力划分

运维最终是需要对外提供服务的，这些服务能力应该由很多底层的自动化平台来承载，个人对其能力划分如下，其实细分到每一层都将发现自动化无处不在，而服务化的能力其实是自动化平台的核心能力，能力划分具体如图 1-6 所示。

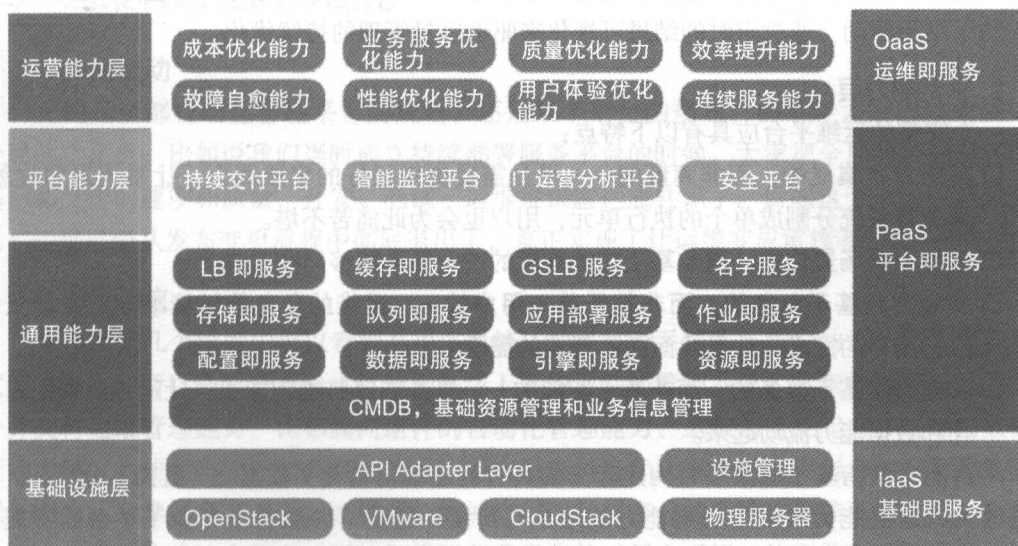


图 1-6 自动化平台

##### 1. 运营能力层

运营能力可体现 IT 运营价值，把 IT 的运营价值和业务场景紧密联系在一起，这些场景和之前所谈的运营价值体系（质量、成本、效率和安全）是一致的。在运维发展的不同阶段，IT 系统的运营价值体现会有所不同，IT 运营的核心方法是要有迭代式的思维。

对于很多企业来说，自动化提升效率是运维的第一个价值突破点；再往后，业务的高可用保证和成本控制，则是下一个价值方向；再之后，精细化运营的业务支撑则是更高的诉求，类似于质量要求（质量的概念非常宽泛）。越往后，越能凸显数据的价值，而非自动化工具的

价值。因此我个人觉得在某一个阶段，自动化平台突破之后，主要的瓶颈将不是效率，而是数据化 IT 运营的能力。该能力在依赖平台的同时，更依赖于运维团队的业务理解能力和经验总结。数据化运营能力是精细化的运营能力，是面向产品的，从底层的基础设施质量、到应用的访问体验、再到产品发布后的用户满意度，等等。

这一层的能力表现为一个具体的**产品形式+运营方法**，从而可以确保能够很好地闭环起来。举个例子来说：基于资源容量管理的成本优化能力，首先需要有一个贴合面向应用的容量分析模型，现实中一般是通过应用所消耗的资源（CPU、内存、I/O、网络等）进行分析运算；其次是需要通过 IT 运营分析产品来呈现应用的容量水平（当前、历史等）；基于可视化的数据呈现，建立相应的容量管理机制。这里又分为如下三点：

（1）建立标准。低负载需要提升资源使用容量、高负载则需要扩容资源（降低资源使用容量）。

（2）明确责任人和职责。确定容量管理的负责人，可以按照应用的粒度进行划分，同时还要明确这部分的管理要求，对于大规模服务来说，可以借用考核的工具来进行驱动。

（3）结果驱动。通过定时的结果可视化来驱动容量管理的持续优化。

## 2. 平台能力层

一个完整的运维平台应具有以下特点：

- ❑ 其能力是**集成的**，而非**离散的**——平台需要提供很好的集成能力，让系统得到收敛，避免将系统分割成单个的执行单元，用户也会为此痛苦不堪。
- ❑ 其能力是**场景化的**，而非**基于功能需求的**——场景能够串联工具。
- ❑ 其能力是**基于角色的**，而非**基于单一用户的**——运维的角色能够清晰地定义场景需求，用户的需求往往是片面而不真实的需求。
- ❑ 其能力是**基于事务的**，而非**基于职能的**——事务能够跨越职能组，让运维组织的自动化和数据能力流动起来。

平台能力是指基于底层平台构建起来的具有的运维自动化/数据化（监控+分析）/安全的能力，这层能力实现了底层能力的组合与封装，屏蔽了底层各个专业子平台的实现细节，是面向业务运维场景的，比如说应用交付、资源交付、业务交付、持续反馈等。

## 3. 通用能力层

通用能力层是基于基础设施之上封装的公共服务能力，这层架构的能力可分成两个部分：一部分是面向业务技术架构的，另一部分是面向运维服务架构的。图 1-6 中所列的服务只是其中的一部分，这个也是我经常和交流者强调的能力建设的核心，不能把这个问题留给下面的资源能力层，也不能交给上层的平台能力层。

对于线上技术架构来说，通用能力层将会涉及名字服务、负载均衡服务、分布式缓存、消息队列、分布式关系存储等，运维需要对其技术实现的工作人员要求 API 直接调用的服务能力。

对于运维服务来说，通用能力层提供了资源服务、作业服务、部署服务、F5 管理、GSLB 等。这层的平台能力我一直将其理解成是 PaaS 平台的核心，有了它们其实就可以实现端到端的能力调度。

该层服务能力平台可以很好地对上层平台进行积木式的支撑，同时还可以对底层设施层能力做服务化能力交付，脱离了资源交付的范畴。

#### 4. 基础设施层

基础设施层是资源交付层，对于一个运维系统来说，应该屏蔽底层基础设施的交付能力，无论是 IaaS，还是物理层基础设施。尤其是对于一些 IaaS 云平台来说，更应该屏蔽 IaaS 底层实现的细节差异，通过 API 服务向上提供能力。国外早年就有了同类的产品，如 RightScale，它很好地实现了多云管理的能力。

## 1.5 运维自动化的方法论

### 1. 全局驱动

无论是全部自动化管理平台的规划，还是某个平台的规划，都希望大家能够找到一个全局的立足点。比如说我们当时成立持续部署服务平台的时候，大家把全局的目标对齐于提高产品交付的速度和质量，开发、测试、运维很快就达成共识了。目前这个平台建设完成之后，运维已经从发布变更流程中彻底退出了，真正实现了让运维变成审核者。

### 2. 分而治之

从上面的几个维度中可以看到有很多系统，如果每个系统都要建设的话，那么周期和难度都将很大。所以需要分而治之，特别是线上架构组件的管理系统，更需要随着组件的交付一并交付运维管理能力，比如面向组件的自动化管理能力、运维的监控能力、运维的数据分析能力等。之前我也表达过类似的观点，所有只交付组件，不交付管理能力的研发都是要流氓。因为从运维的角度来说，这样低价值的交付产品越多，越会导致运维不堪重负。而如果让运维从头去构建这个管理，则他们需要花费很多的时间去了解，从而导致系统建设周期拉长。举个例子，比如说某个分布式 cache 服务，做得不好的，是通过读取日志然后对其进行监控；做得好的，是给你开启一个管理端口，让你从端口中读取状态信息。这就大大降低了系统的复杂度（不用进行日志采集和处理组件了）。

分而治之，其实就是让不同的团队做不同的事情，不要将所有事情全部压给运维；其次不同的时期建设不同的系统，不要在同一时刻做很多系统，从而避免战线过长。当然如果有很多运维研发人员的话，就另当别论了。

### 3. 自底向上

自底向上，其实是让大家找到一个更清晰更具体的系统建设目标来展开工作。从系统分

解上，来让大家规避被一个庞大而模糊的目标带入歧途。如果一上来，我们就说要做一个全自动的运维管理系统，那样很容易就会让运维研发团队迷失方向。所以这里可以先设定全局和最终目标（全自动化），然后从底层逐步构建地基，做框架，最后再盖一个完整的房子，详见图 1-1。

#### 4. 边界清晰

边界有两个维度，一个是管理边界；一个是职能边界。

首先是管理边界，其是从 Owner 的角度出发的，谁产生服务，谁就是 Owner，管理统一都是运维。比如研发提供了一个统一的分布式消息队列服务，那么 Owner 就是研发，他应该对可运维性负第一责任，不要让运维去承担这个服务的 WebAdmin 管理系统建设任务。

其次是职能边界，深层次的理解是组件的功能范围。对运维架构师的考验也就在这儿，比如说让 LVS 去承担业务异常的容灾和容错切换是不合适的；让 DNS 跨过 LVS 层，负责后端服务异常的自动容错处理也是不合适的。如果不把职能界定清楚，将会导致系统做很多无用功，这会增加系统建设的复杂度。

#### 5. 插件化

插件化的思维无处不在，在面对纷繁复杂的管理对象时，我们进行抽象，提供管理模式，然后将具体的实现交给用户，这点在我们日常所见的运维系统中经常可以看到，比如说 Nagios 就是一种插件化的采集思路。对于配置管理来说，Puppet 采用的也是这个思路。对于最上层的调度管理系统，可以让运维自己去编写执行器，特别是和业务紧密相关的，但最终运维整个控制权还是要交给平台。我的经验是，在应用服务层和架构服务层，不要引入插件化的管理方案，过多的插件化部署，会让生产环境的管理最终混乱不堪，甚至失控。所以提供类 SSH 界面的运维发布和部署平台，是没有任何运维价值的。

## 1.6 运维自动化系统的实现

挑战自动化的极致场景（可视化），是运维人员对极致的追求。极致的自动化是运维事务全流程的自动化，运维事务全流程自动化是包含了一次应用完整交付所涉及的所有资源的自动化能力，比如说 DNS 资源、负载均衡资源、数据库资源、服务器资源、配置资源等。下面将列举几个典型的运维自动化系统以供大家参考。

### 1.6.1 DNS 管理系统

DNS 是 Web 形态下的一个重要入口，用户服务的访问严格依赖于这个服务入口。现在一般被称为 GSLB（全局服务负载均衡调度），目前是 CDN 服务中的重要服务节点。实现的目标都是要解决运维从哪里来，到哪里去最快，当目标机房发生故障的时候，如何把服务调度走。



在移动 APP 大量应用的今天, DNS 协议的缺点已经逐渐暴露出来了, DNS 解析时间长, 另外还经常会被劫持。因为有端的控制, 现在逐渐开始走 HTTPDNS 的服务, 通过 HTTP 服务的方式获取域名对应的 IP 地址, 此时由 DNS 平台直接对外提供 HTTP 服务。在有端 App 的情况下, 还可以借助端的数据挖掘技术, 识别非权威 DNS 域名是否存在被劫持的情况。系统需要保持和业务的与时俱进。

这里还需要注意一个问题, 内部 DNS 能否统一管理? 理论上是可以的, 把单个机房当作单个的 view, 不过我不建议将两个场景耦合在一起, 尽管这样能够实现统一管理。

系统 Demo 如图 1-7 所示。

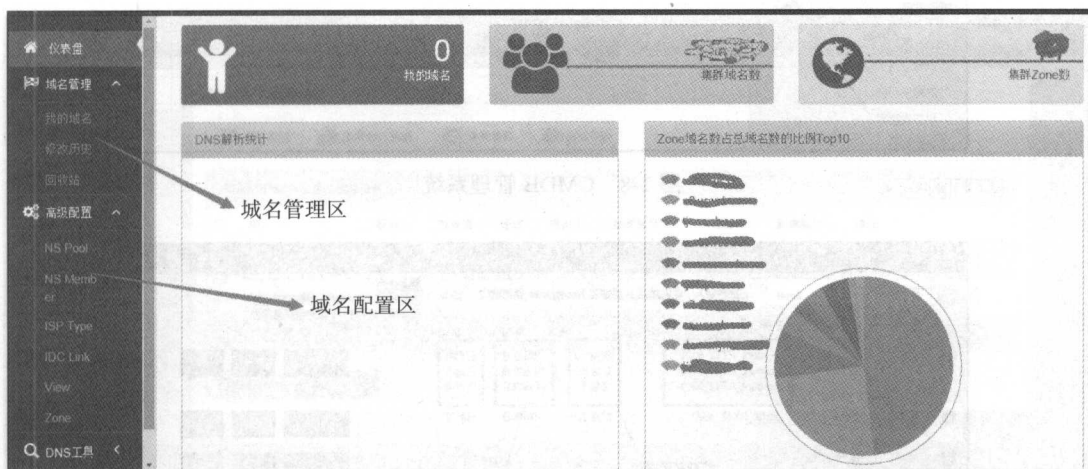


图 1-7 DNS 管理系统

## 1.6.2 CMDB 管理系统

CMDB 管理系统的建设这里就不展开介绍了, 感兴趣的读者可以关注微信公众号“互联网运维杂谈”, 并参阅《运维平台之 CMDB 系统建设》一文。

系统 Demo 如图 1-8 所示。

## 1.6.3 名字服务中心系统

“名字服务中心系统”的概念最初来自于 Zookeeper, 该系统结合实际情况, 实现了名字服务中心。把程序接口之间的调用抽象成单个服务之间的调用, 在服务中心实现调度的统一注册、鉴权、ACL、容灾容错控制。将其看作线上服务最核心的系统, 一点也不为过, 并且它还是收益最大的系统, 可直接替换掉 DNS、LVS, 降低线上系统对运维系统的依赖性。

系统 Demo 如图 1-9 所示。



图 1-8 CMDB 管理系统

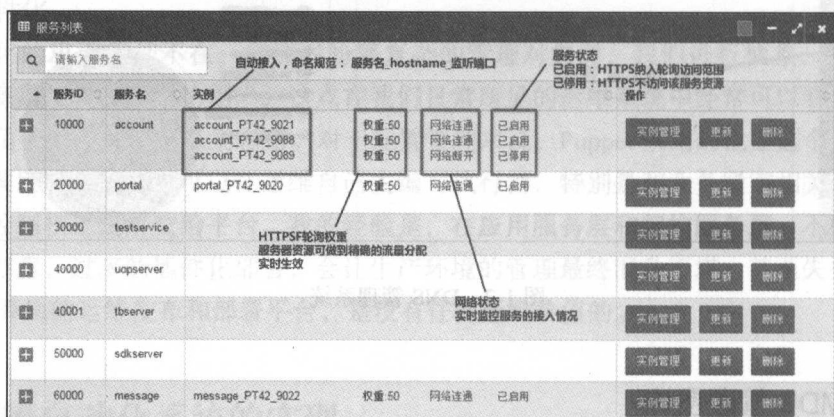


图 1-9 名字服务中心系统

### 1.6.4 持续部署管理系统

持续部署是应用升级的核心系统，该系统每个月都承担着大量的变更。在系统规划之初，我们就给它设定了清晰的业务管理目标：持续交付的一部分，实现图 1-10 中的 4 个维度管理目标；也设定了具体业务的运维目标：升级所有的包和配置，且让业务运维彻底退出业务的变更流程。具体如图 1-10 所示。

系统 Demo 如图 1-11 所示。

持续部署系统是持续交付系统的核心（持续集成、持续测试、持续部署、持续反馈），它是产品发布到达生产环境的关键步骤。在这个平台的建设上，运维人员应该将它作为突破的

第一个点。在该平台搭建完成之后，运维就可以从日常的部署事务中解放出来了。

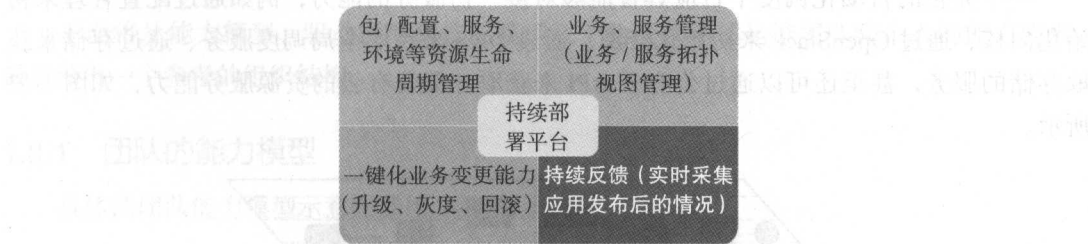


图 1-10 持续部署管理系统示意图

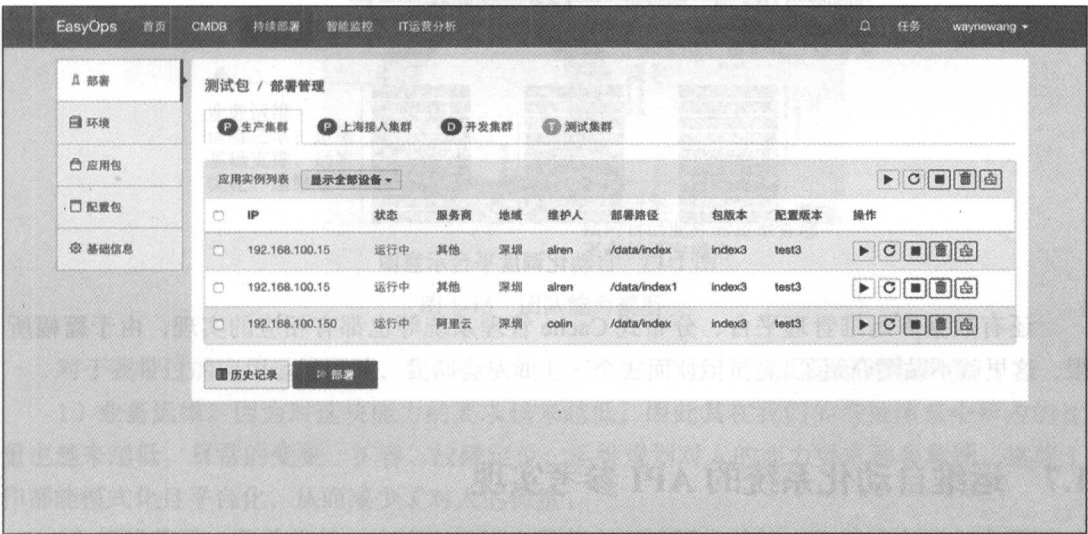


图 1-11 发布系统

### 1.6.5 运维调度管理系统

运维调度平台又称为调度编排系统，编排是一种场景化的运维能力封装，是对复杂运维事务的封装。我们在平时的运维过程中能够看到很多复杂的运维场景，比如说容灾切换、故障处理、服务迁移等。这些场景，很多时候都不是单一的动作就能够完成的，往往需要借助多种运维能力组合，如图 1-12 所示。

在图 1-12 中，我们把 Ops 自动化调度下面的服务支撑层分解为三部分：工具平台 OpsStore，用来编写日常的运维工具；外部服务，用于公共

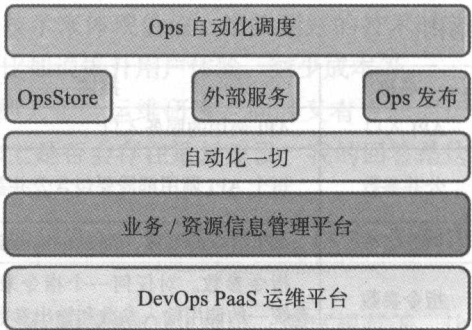


图 1-12 运维调度管理系统示意图



API 对外提供封装；Ops 发布，用于提供代码持续部署服务。

一个完整的自动化调度平台应具备能够对接一切服务的能力，例如通过配置管理来初始化内核、通过 OpenStack 来初始化资源、通过 DNS 来获取全局调度服务、通过存储来获取存储的服务，甚至还可以通过公有云 API 来获取外部公有云的资源服务能力，如图 1-13 所示。

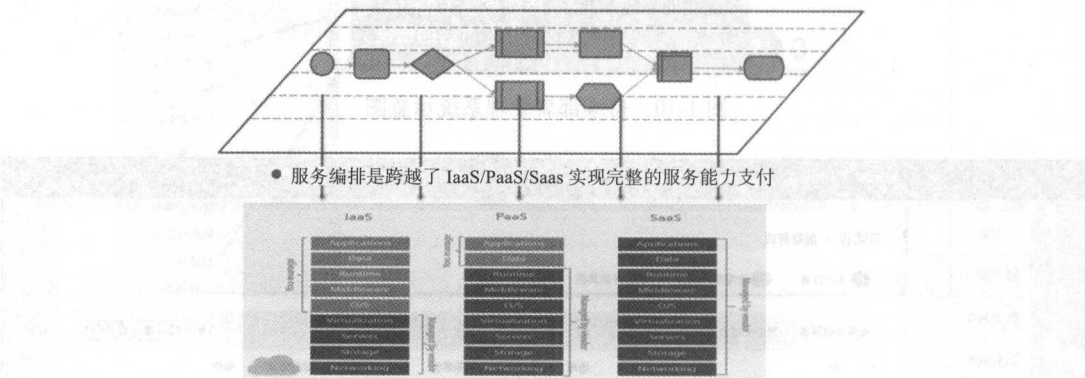


图 1-13 自动化调度平台示意图

还有数据库运维管理平台、分布式 Cache 管理系统等也都有相应的实现，由于篇幅所限，这里就不贴图介绍了。

## 1.7 运维自动化系统的 API 参考实现

所有的底层系统都是通过 API 对外提供服务的，API 可供各个系统使用。接口的使用需要通过授权来获得，建议这个授权可以是基于系统级别的，也可以是接口级别的，而不是采用统一开放的模式。另外接口内需要有相应的一些权限控制，以避免底层服务被任意操作。

可以仿照 AWS 的接口实现方式，统一实现 API 的接口开放访问地址，同时统一协议（HTTP、HTTPS），协议可以使用 Get 的方式进行访问。图 1-14 所示是一个开放 API 的结构。

名称	描述	示例
API 入口	API 调用的服务入口	http://auto.**.com/
公共参数	每个 API 调用都需要包含公共参数	包含了颁发的 access_id、时间戳、API 版本、签名、签名的方法 (sha1、md5) 等
指令名称	API 指令的名称，例如 newrdsinstance 等	每个系统都需要注册统一的服务名到服务中心
指令参数	指令参数。对任何一个指令来说，都应该有统一的调用输入参数和输出参数的说明	在界面化的 API 中心里有统一的在线说明手册

图 1-14 开放 API 的结构

## 1.8 运维自动化依赖的团队模型

下面将从能力模型、驱动模型和技能模型三个角度来阐述运维团队和个人的能力要求，最后给出一个参考的组织结构。

### 1.8.1 团队的能力模型

具体的团队能力模型示意图如图 1-15 所示。

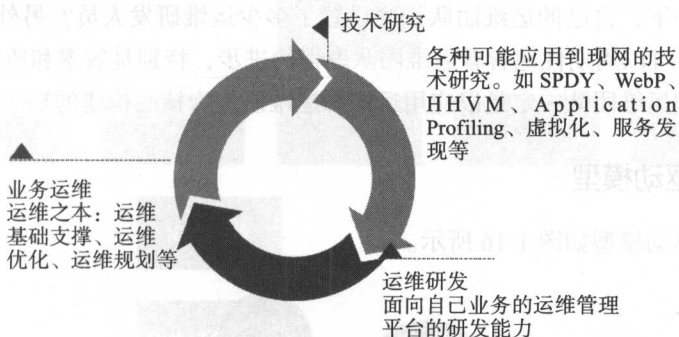


图 1-15 团队能力模型

对于我带过的应用运维团队，我都会从如上三个方面对组员提出运维能力要求。

1) **业务运维**。因为对这块能力的要求越来越低，因此其在我们的考核体系中所占的比重也越来越低。日常的变更、扩容、故障定位、运维规划对人的能力要求都非常低，这些工作都能模式化且平台化，从而减少了对人的倚重。

2) **运维研发**。我希望每一个应用运维人员都有运维研发的能力，但这在现实中是不可能的。对于应用运维团队和运维部门来说，运维研发的配备必不可少。在应用运维团队的内部，可以让有研发能力的人迅速承担面向业务运维平台的建设，或者参与到部门的运维系统建设中，可以抽出 50% 的时间参与研发。运维研发能力是能够让团队价值迅速达成的有效保证，没有研发能力的运维不能成为一个好运维。

3) **技术研究**。运维是一个技术团队，需要通过技术来体现价值，当找到好的技术时就要想着如何将技术应用到业务上，为用户带来价值，比如说提升用户体验，减少成本等。

这个时候就会产生一个问题，应用运维团队内的人也会运维研发，同时又有专职的运维研发团队，那么他们的职责分工如何解决，在工作上是否会存在重复建设？我的回答是这样的：

首先，可以把运维研发初期定位在公共服务平台的研发上，比如说 DNS、LVS、配置管理、监控系统、CMDB、数据分析平台等。

其次，运维研发还需要制定相应的运维研发规范，代码规范、UI 规范、测试规范等，让所有参与运维研发的人统一遵守，包括应用运维研发的组员。

最后来说一下应用运维小组内的研发能力该如何发挥的问题。其实在很多运维团队中，运维都是跟随业务的，一则可以让应用运维研发人员开发面向业务的运维系统，因为他们最了解该业务的需求，能够实现自己想要的；另外一种更好的操作方式，是让应用运维小组内的研发人员抽出 50% 的时间参与到以运维研发牵头成立的虚拟研发小组中。一则可以进一步提高应用运维的研发水平；另外还可以提高运维研发对业务运维的理解，同时还能提高带队作战的能力。

那么，运维研发和应用运维的比例应该设置成多少比较合适？我个人认为 3:1 比较合适，大家也可以自检一下，自己的运维团队到底设置了多少运维研发人员？另外想要检测运维研发配备是否足够，可以周期性地看看运维团队取得的进步，特别是效率和质量等维度。

一个高性能的运维团队一定是以应用运维和运维研发为核心构建的！

## 1.8.2 团队的驱动模型

具体的团队驱动模型如图 1-16 所示。

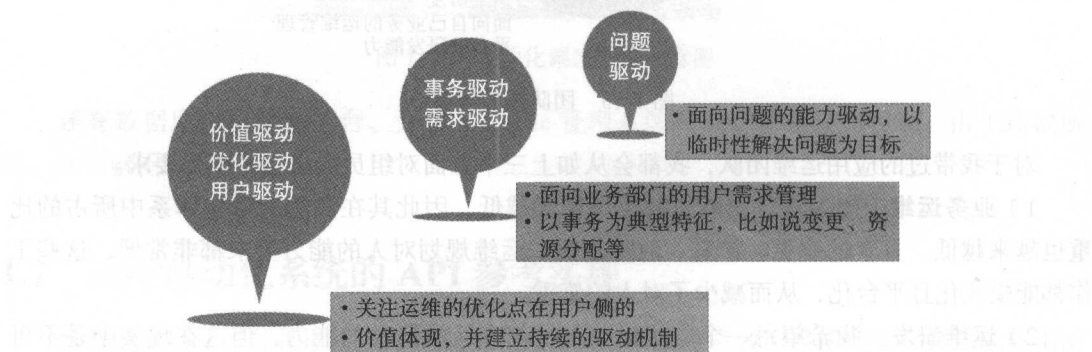


图 1-16 团队驱动模型

团队的驱动力不同，带来结果的就会完全不同。为什么很多运维人员都说自己很辛苦？这时你可以思考一下到底是什么在引导着你进行运维工作？传统的维护，往往都集中在第一阶段和第二阶段，而进入到高阶运维体系之后，我们需要迅速切换到价值驱动和用户驱动的维度上来。有了用户驱动和价值驱动，对运维的效率和质量就都会有更高的要求，对于外部驱动我们必须走自动化和平台这条道路。建议大家在平时的工作中加入质量、效率、成本等一些 KPI 要求，不要只局限于自己所做的事情，而是要关注自己所做的事情对产品 and 用户的影响。

## 1.8.3 团队的技术模型

BAT（百度、阿里、腾讯）很早就实施了职业通道体系，在运维侧细分了多个能力通道，比如说网络运维、业务运维、运维研发、DBA 等。对于运维人员的成长也有明确的要求和衡

量体系，在此我就不详细介绍了。

1.8.4 参考的运维团队组织结构

我们不一定要按照这个结构明确设置运维小组，但是运维的职能差不多就是这样。我还有另外一个建议，最好将公共服务研发团队和运维团队放在一个组织结构下，这样将会有利于公共化服务的推广，而公共化服务对运维效率的影响是最大的，如图 1-17 所示。



图 1-17 运维团队组织结构

至此，自动化平台的深度解码已经完成。本章从多个层面带领大家了解运维自动化，其实还是希望能给大家带来一点借鉴意义。大胆地往前走吧，一切都有可能，唯独那些实现不了的，都是我们人的问题，无它。

1.9 小结

无论是传统企业还是中小型互联网公司，运维自动化都是当前运维阶段的核心能力。实现了运维自动化，传统的人肉运维状况才会被改变，才会有更多的精力去思考更大的运维价值。在建设运维自动化平台的过程中，对运维自动化的需求进行清晰的识别、界定、规划和落地是很考验运维研发者能力的事情，本章从整体规划入手，给出了一些具体的方法论，同时还给出了几个实现的例子，目的是帮助大家看到一条清晰的建设路径。

## 利用 Facter 和 Django 快速构建 CMDB

### 作者简介

陈立军，金山西山居 DevOps，原新浪研发系统开发。

刘宇，网名守住每一天，金山西山居架构师，《Puppet 实战》一书作者，《Puppet 实战手册》译者之一，《Python 高级进阶》译者之一，InfoQ 社区编辑，自动化运维专家。

CMDB (Configuration Management Database)，又称配置管理数据库，更多的时候我们习惯将其称为资产管理系统。它既是 ITIL 标准体系的核心，又是运维的基础核心系统。它通常位于整个运维自动架构的底层，但在运维自动化体系中，它又起到了极其关键性的作用。据我个人了解目前很多中小型企业，乃至一些中大型互联网公司，都还在采用传统的 Excel 来管理资产。

采用 Excel 维护资产时，需要耗费大量的人力和精力，这些工作包括：数据的采集、整合、记录、维护、检验和更新，每一项都比较繁琐。而采用开源 CMDB 系统却又不夠灵活，无法和公司的其他系统相结合。面对这些问题，自行研发一套适用于公司的 CMDB 系统，是当务之急。

本章将会详细讲解，如何利用开源软件 Facter 和 Django 快速构建一个小型的 CMDB 系统。界面与功能不算复杂，这也是它轻量的体现，本章更多的是提供一种思路，以达到抛砖引玉的作用，仅供大家参考。



说明

本章的所有代码都托管在 Github 网站上 <https://github.com/oysterclub/open-cmdb>。其中脚本集中在 CMDB 目录中。

## 2.1 CMDB 简介

CMDB 用于存储和管理企业 IT 架构中各种设备的配置信息，其中包括主机、项目、用户、机房、网络等。其被认为是 ITIL 服务管理的核心，所有流程所需要使用的配置信息都将通过 CMDB 来进行获取，例如监控、Dashboard、自动化、流程等。CMDB 在企业 IT 架构中的核心地位如图 2-1 所示。

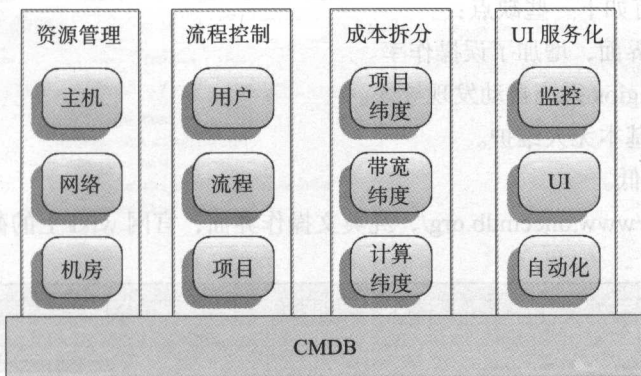


图 2-1 CMDB 的核心地位

CMDB 特性如下：

- ❑ 设备信息统一化、标准化。
- ❑ 设备信息可维护性更强。
- ❑ 设备信息关系清晰化、可视化。
- ❑ 设备信息的查询、更新更快。

当然，在实际生产中，会对上述特性和涵盖范围有轻重大小的区分，为了能够更加合理、更加准确地设计出适合自有业务模式的 CMDB，下面我们来看下几大主流开源 CMDB 是如何设计的。

## 2.2 开源 CMDB 介绍

目前主流的开源 CMDB 软件包括：OneCMDB、CMDBuild、Itop CMDB、Rapid OSS、ECDB、i-doit 等，其中比较出名的是前二者，因此本节将重点对比这两款开源软件。

### 2.2.1 OneCMDB 介绍

OneCMDB 主要面向的是中小型企业。可以作为一个独立的 CMDB 来保持软件和硬件资产及其相互关系的轨道。由于其具有开放的 API，因此其也可以是拥有灵活的强大的配置管理引擎的其他服务管理软件。



OneCMDB 易于安装和填充数据，它有一个无需用户具有编程能力就能改变和增强的数据模型，它能让用户轻松做到如下几点：

- ❑ 创建 CMDB 数据模型，而无需写代码。
- ❑ 填充数据，可以通过网络自动发现。
- ❑ 通过各种灵活的导入和转换机制来从外部源获取信息。
- ❑ 导入 / 导出网络配置信息从 / 到 Nagios 网络监控系统。

OneCMDB 也有如下一些缺点：

- ❑ 纯英文操作界面，增加了误操作率。
- ❑ 只支持从 Nagios 系统自动发现导入。
- ❑ 现在该产品基本无人维护。
- ❑ UI 可定制化低。

官网地址 <http://www.onecmdb.org/>，纯英文操作界面，官网 wiki 上的截图如图 2-2 所示。

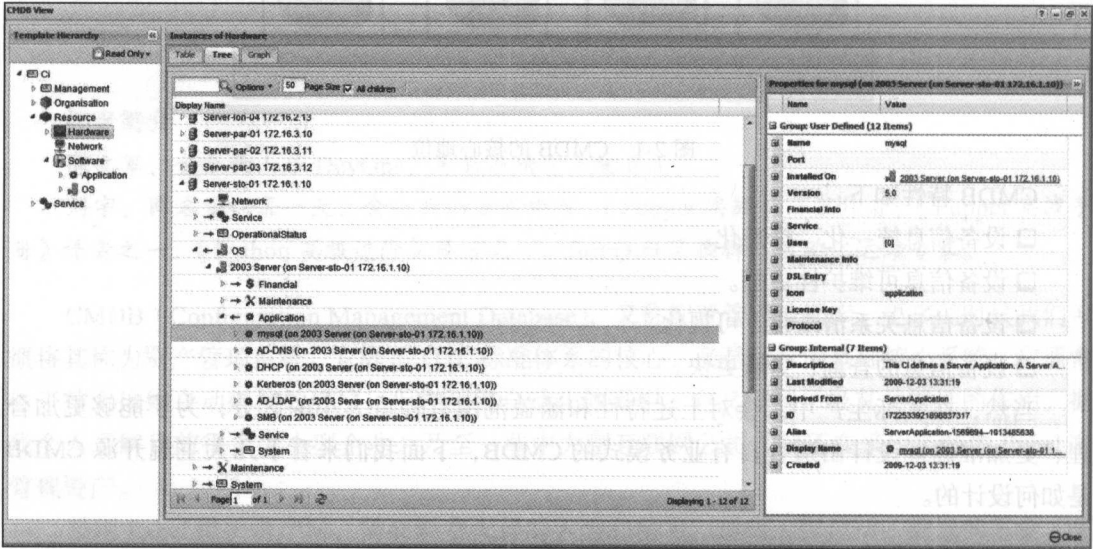


图 2-2 OneCMDB 操作界面

2.2.2 CMDBuild 介绍

CMDBuild 是一个通过 Web 界面配置的 CMDB 系统。可以通过 Web 界面来进行建模、创建资产数据库，并处理相关的工作流程。

CMDBuild 可用于集中管理数据库模块和外部应用：自动库存、文档管理、文本处理、目录服务、电子邮件、监控系统、用户网站、其他信息系统等。

官网地址 <http://www.cmdbuild.org/>，也是纯英文操作界面，截图如图 2-3 所示。



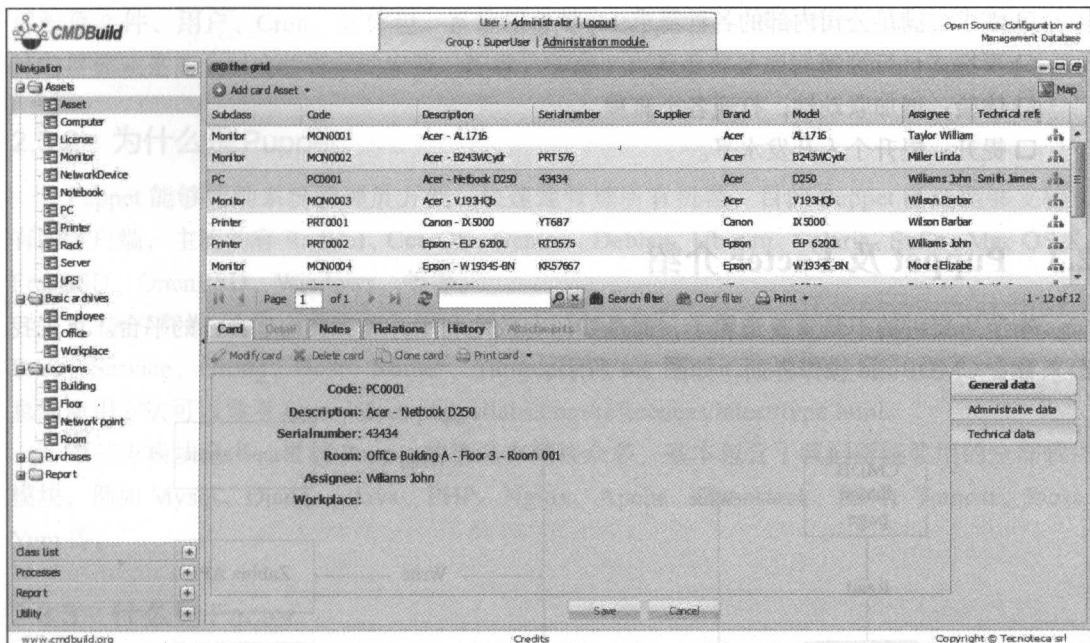


图 2-3 CMDBuild 操作界面

CMDBuild 应该是使用者比较多的一款产品，它具有如下优点：

- ❑ Ajax 操作十分便捷（采用了 ExtJS 作为支持）。
- ❑ 数据格式自由定制（在 GLPI 中，资产的数据格式都已经定义好了，用户很难再做修改）。
- ❑ 支持多种开源标准（XPDL）。
- ❑ 可以自定义 Workflow，便于 ITIL。
- ❑ 有专门的团队在不断进行维护，截至本章写作时，最新版本为于 2016 年 6 月 16 日更新的 2.4.1 版本。
- ❑ SOAP 和 REST 的 Webservice 接口。

要说其缺点也就只有一条让人望而却步：文档少、资料极少。

### 2.2.3 其他的开源 CMDB

随着开源潮流的发展壮大，开源的 CMDB 也越来越多，比如 Itop CMDB、Rapid OSS、ECDB、i-doit 等。然而这些开源的样式长得都差不多。最大的弊端在于不能有效地与其他系统友好结合。如果只是单纯地进行统计使用，也未尝不是一种选择。有时，现有的开源工具和系统不能满足业务发展的需求，我们就需要修改或完全自行编写一个符合业务需求的工具或系统，即“造轮子”。

自己造轮子有如下几个优点。

- ❑需求：满足公司内部的各种需求。
- ❑可控：自行控制。
- ❑体验：增加联动性，打通各个流程。
- ❑提升：提升个人开发水平。

### 2.3 Puppet 及 Facter 介绍

本节所要讲的不是重复造轮子，而是如何有效地利用开源工具构建自己的平台。首先我们来看看 CMDBuild 操作界面，如图 2-4 所示。

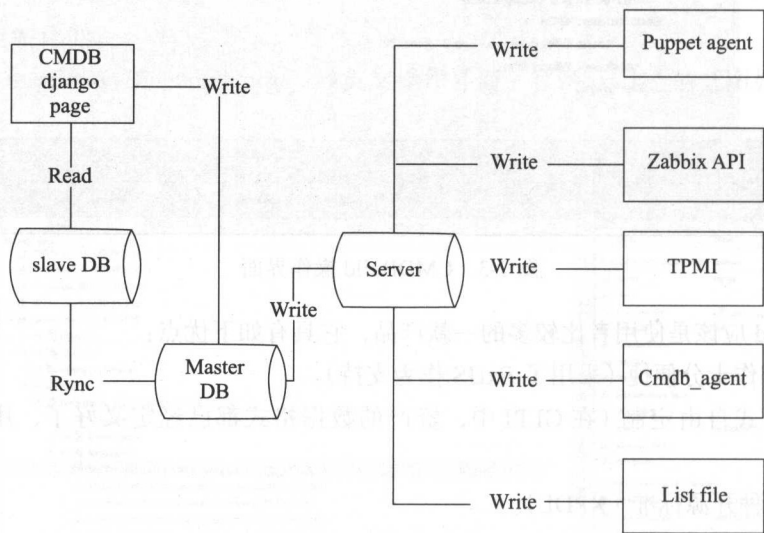


图 2-4 CMDBuild 操作界面

从图 2-4 中可以看出，CMDB 的信息收集可以是多种多样、共融共存的：

- (1) 通过一些 Agent 客户端收集信息，然后注册到中心服务器数据库。
- (2) 通过监控系统收集，比如 Zabbix、Nagios、IPMI 等。
- (3) 通过配置管理工具收集，比 Puppet、SaltStack、Ansible 等。

我在本文中使用配置管理工具 Puppet 的 Facts 来进行信息收集，最主要的原因是，目前我使用的是 Puppet 来管理所有的机器，因此使用 Facts 最方便、简单、快捷，成本低而且效率高，如果读者使用的是 Ansible 或其他配置管理工具来做管理，本文中所介绍的方法也可以通用。

#### 2.3.1 什么是 Puppet

通常定义：Puppet 是一个跨平台的集中化配置管理系统，它使用自身的描述语言，可管

理配置文件、用户、Cron、软件包、系统服务等，Puppet 把这些统称为“资源”。Puppet 设计的目标就是简化对这些资源的安装、配置、管理，以及妥善处理资源之间的依赖关系。

## 2.3.2 为什么是 Puppet

Puppet 能够帮助系统管理员方便、快速地管理所有机器，目前 Puppet 已经能够支持所有的客户端，主流的有 RedHat、CentOS、Gentoo、Debian、Ubuntu、Solaris、SuSe、Mac OS X、FreeBSD、OpenBSD、Windows，等等。

支持的资源众多：目前 Puppet 支持的资源有很多，其中常用的包括 File、Package、Exec、Service、Group、Host、Router、Yumrepo、User、Cron、SSHKey 等，更多的相关信息和使用方法可以参考 <https://docs.puppetlabs.com/references/latest/type.html>。

第三方模块众多：目前 Puppet 的第三方模块众多，基本包含了我们所能使用的全部软件模块，例如 Mysql、Django、Java、PHP、Nginx、Apache、Openstack、SSH、Tomcat、Jboss、Yum 等。

## 2.3.3 什么是 Factor

Factor 是 Puppet 跨平台的系统性能分析库。它能发现并报告每个节点的信息，在 Puppet 代码中是以变量的形式出现的。它返回的是每个 Agent 的 fact 信息，这些信息包括主机名、IP 地址、操作系统、内存大小及其他的系统配置选项，这些 fact 信息在 Puppet Agent 运行的时候进行收集并传递给 Master，同时被自动创建为可以被 Puppet 使用的变量。

## 2.3.4 Factor 的特点

Factor 最大的作用就是收集服务器系统信息，包括主机名、IP 地址、操作系统、内存大小及其他的系统配置选项。这些系统配置选项正是 CMDB 所需要的基础核心数据。

我们先来看看 Factor 收集的系統数据，下面将列举一些常用的系统数据，具体信息如下。

(1) Factor 获取 fqdn 信息，在 Factor 中 fqdn=hostname + domain:

```
$ factor fqdn
puppet.domain.com
```

(2) Factor 获取 IP 地址:

```
$ factor ipaddress
10.20.122.100
```

(3) Factor 获取 MAC 地址:

```
$ factor macaddress
00:1A:4A:25:E2:10
```

(4) Factor 获取空闲内存大小:

```
$ factor memoryfree
1.61G
```

(5) Factor 获取内存大小:

```
$ factor memorysize
1.83G
```

(6) Factor 获取操作系统:

```
$ factor operatingssystem
CentOS
```

(7) Factor 获取 CPU 信息:

```
$ factor processors
{"models"=>["Intel Core 2 Duo P9xxx (Penryn Class Core 2)", "Intel Core 2 Duo P9xxx (Penryn Class Core 2)"], "physicalcount"=>2, "count"=>2}
```

(8) Factor 获取机器运行时间:

```
$ factor uptime
1 day
```

2.3.5 Factor 变量

Factor 目前的最新版本为 3.4.1 (截至本章写作时), 支持的变量有 131 个, 所支持的变量可以在官方网站 [http://docs.puppetlabs.com/facter/latest/core\\_facts.html](http://docs.puppetlabs.com/facter/latest/core_facts.html) 中查看。Factor 所支持的变量可以按使用频率简单划分为常用类型和不常用类型, 具体如图 2-5 所示。

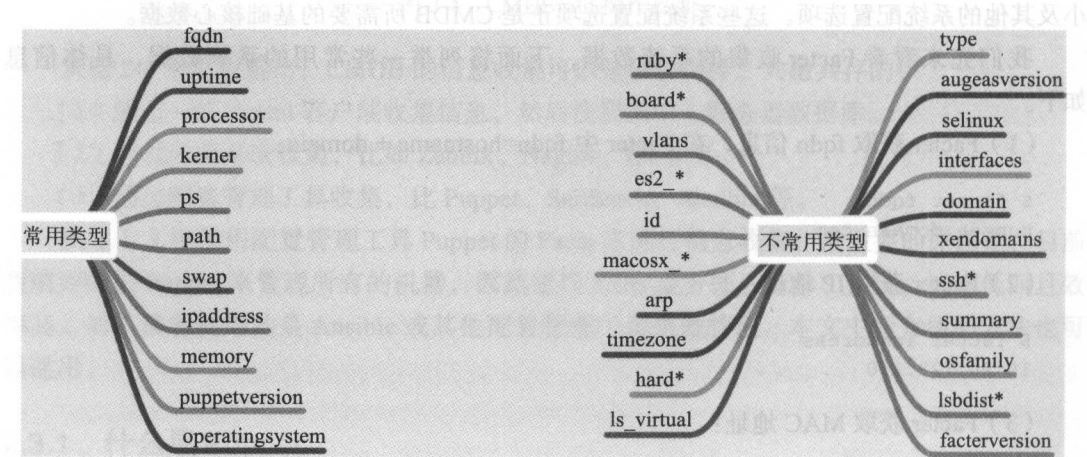


图 2-5 Factor 变量分类图

## 2.4 如何利用 Python 获取 Facts

### 2.4.1 工作原理

通过 2.3 节的学习可以知道 Facts 可以获得主机的系统信息，并以 K-V 形式进行存储，我们只需要处理 Puppet Server 收集的 Agent Facts 信息、入库，然后通过 Django 来读取数据库信息即可，如图 2-6 所示。

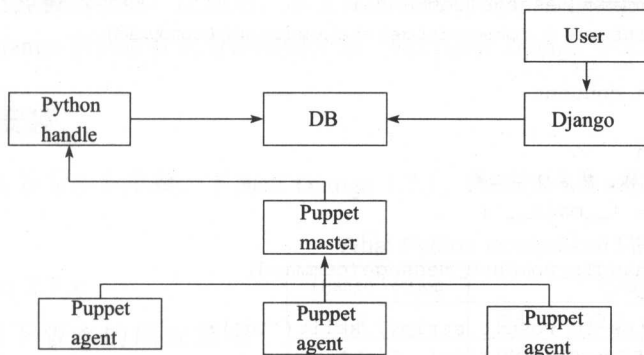


图 2-6 Facts 信息获取流程图

### 2.4.2 利用 Python 脚本获取 Facts

首先要确定系统中已经安装了 Puppet，如果没有，可以从 <http://yum.puppetlabs.com/> 下载，并参考 [https://docs.puppetlabs.com/puppet/latest/reference/install\\_pre.html](https://docs.puppetlabs.com/puppet/latest/reference/install_pre.html) 进行安装，接下来看看如何通过 Python 程序来获取 Facts 信息（注意：下面的程序是查看当前机器的 fact 信息，下面的这个程序对结果不会做过多的处理，在后面进行 CMDB 项目的时候将详细讲解 fact 的数据处理），实例程序 `facter_message.py` 的内容如下：

```
#!/usr/bin/python
# encoding: utf-8
__authors__ = ['LiuYu']
__version__ = 1.0
__date__ = '2015-08-19 14:34:44'
__licence__ = 'GPL licence'
```

# 导入模块

```
import commands
```

```
import re
```

# 定义一个变量

```
command = 'facter'
```

# 定义要打印的列表

```
show_list = [('fqdn', u'主机名'),
```

```
              ('domain', u'域名'),
```

```
              ('uptime', u'运行时间'),
```

```

        ('operatingsystem', u' 系统 '),
        ('kernelrelease', u' 内核版本 '),
        ('ipaddress', u' IP '),
        ('macaddress', u' MAC '),
        ('memorysize_mb', u' 内存 MB '),
        ('processors', u' CPU '),
        ('blockdevices', u' 磁盘 '),
    ]
# 定义一个处理命令的函数
def handle_command_message(command):
    status, content = commands.getstatusoutput(command)
    if status == 0:
        return content
    else:
        return
# 通过函数处理名称，然后打印结果
if __name__ == '__main__':
    result_dict = {}
    result = handle_command_message(command)
    if result:
        for line in result.strip().split('\n'):
            if re.findall('=>', line):
                key, value = line.split('=>', 1)
                result_dict[key.strip()] = value.strip()
        for f_k, f_s in show_list:
            if f_k in result_dict:
                print f_s, ': ', result_dict[f_k]

```

下面我们来运行 `facter_message.py` 程序，打印结果：

```

$ python facter_message.py
主机名 : puppetclient.domain.com
域名 : domain.com
运行时间 : 1 day
系统 : CentOS
内核版本 : 2.6.32-431.1.2.0.1.el6.x86_64
IP: 10.20.122.111
MAC : 00:22:E2:5E:4D:10
内存 MB : 996.48
CPU : {"count"=>1, "models"=>["QEMU Virtual CPU version 1.1.2"], "
磁盘 : sr0,vda,vdb,vdc

```

通过如上的简单代码就可以将 Facts 的信息进行集中处理。

## 2.5 使用 Django 快速构建 CMDB 系统

### 2.5.1 Django 介绍

Django 是一个免费的、开源的 Web 框架，由 Python 语言编写，由于其是在一个快节



奏的新闻编译室环境中开发出来的，因此它的设计目的是让普通开发者的工作变得简单。Django 遵循模型 - 视图 - 控制器（MVC）框架模式，目前由一个非盈利的独立组织的软件基金会（DSF）维持。

Django 鼓励快速开发和干净实用的设计。Django 可以更容易更快速地构建更好的 Web 应用程序。它是由经验丰富的开发人员来创建的，省去了 Web 开发的很多麻烦，因此你可以专注地开发应用程序而不需要去白费力气地重复工作。

Django 目前已经被运维圈广泛使用，本文在此不会详细介绍 Django 的基础知识，有兴趣的朋友可以去 Django 官网查看更为详细的介绍，同时也有 Django 中文文档可供学习。

## 2.5.2 Django 安装

Django 的安装分为 4 个步骤，下面以 Django 1.7.1、CentOS 6.5 x86\_64 为例进行讲解，详细步骤如下。

### 1. 安装 Python 2.7.x

用 CentOS 7 以下版本的朋友需要将 Python 升级到 2.7.x 以上，Django 对 Python 版本存在依赖，具体如图 2-7 所示。

编译步骤如下：

```
# yum install -y zlib-dev
openssl-devel sqlite-devel bzip2-devel
# wget http://www.python.org/ftp/python/2.7.6/Python-2.7.6.tgz
# tar xzf Python-2.7.6.tgz
# cd Python-2.7.6
# ./configure --prefix=/usr/local
# make && make altinstall
```

（1）安装 easy\_install 工具，操作命令如下：

```
$ wget https://bootstrap.pypa.io/ez_setup.py -O - | python
```

（2）安装 Django，使用 easy\_install 来安装，安装的版本为 1.7.1，具体命令如下：

```
$ easy_install django==1.7.1
```

（3）测试 Django 的安装，操作命令如下：

```
$ easy_install django==1.7.1
$ django-admin -version
1.7.1
```

### 2. MySQL 安装

本文推荐使用 yum 命令进行安装，并设置 MySQL root 密码，创建 cmdbtest 数据库，具体安装步骤如下：

What Python version can I use with Django?

Django version	Python versions
1.4	2.5, 2.6, 2.7
1.5	2.6, 2.7 and 3.2, 3.3 (experimental)
1.6	2.6, 2.7 and 3.2, 3.3
1.7, 1.8	2.7 and 3.2, 3.3, 3.4

图 2-7 Django 与 Python 版本的依赖关系

```
$ yum -y install mysql mysql-server
$ mysql_install_db --user=mysql
$ /etc/init.d/mysqld start
$ mysqladmin -u root password 'cmdbtest'
$ mysql -u root -pcmbdtest -e 'create database if not exists cmdbtest'
```

### 2.5.3 Django 常用命令

完成 Django 的安装后，可以通过如下命令快速熟悉 Django 的操作，以便快速创建一个 CMDB App，如果你对如下这些 Django 命令很熟悉，可以直接跳过。

#### (1) 新建一个 django-project:

```
$ django-admin startproject project-name
```

#### (2) 新建 App:

```
$ django-admin startapp app-name
```

#### (3) 同步数据库:

```
$ python manage.py syncdb
```

#### (4) 启动 Django 服务器:

```
$ python manage.py runserver
```

#### (5) Django Shell 调试:

```
$ python manage.py shell
```

#### (6) 帮助:

```
$ django-admin --help
$ python manage.py --help
```

### 2.5.4 Django 的配置

#### 1. 环境准备

笔者准备了两台测试机器用来进行代码测试，测试机器的环境信息分别如下。

##### (1) 服务端机器信息:

```
IP: 10.20.122.100
Role: puppet server + cmdb
System OS: CentOS release 6.5 x86_64
Python version: 2.7.8 Django version: 1.7.1
Mysql version: 5.1.73
```

##### (2) 客户端机器信息:

```
IP: 10.20.122.111
```

```
Role: puppet agent
System OS: CentOS release 6.5 x86_64
```

## 2. 软件安装

前几节已经对所需要的环境进行了安装，在这里我们再回顾一下：

- (1) master 安装 Puppet Server。
- (2) master 安装 Python。
- (3) master 安装 MySQL。
- (4) master 安装 Django。
- (5) master 安装项目依赖的 Python 模块。
- (6) Agent 安装 Puppet Agent。

## 3. 创建 CMDB 项目

创建 CMDB 项目的同时，在这个项目中创建一个 CMDB App，登录 10.20.122.100，运行如下命令。

- (1) 创建一个 Project：

```
$ django-admin startproject myproject
```

- (2) 进入 myproject 目录：

```
$ cd myproject
```

- (3) 创建一个 CMDB App：

```
$ django-admin startapp cmdb
```

- (4) 创建一个存放静态文件和模板的目录：

```
$ mkdir static templates
```

运行成功后使用 ll 命令就可以看到如图 2-8 所示的目录结构。

## 4. 配置 CMDB 项目信息

在图 2-8 中我们可以在 myproject 目录下看到 settings.py 的全局配置文件，Django 在运行时默认先加载此配置文件，因此我们需要先对它进行定义，需要配置如下 6 个地方，操作步骤具体如下。

- (1) 修改数据库设置：

```
DATABASES = {
    'default': { 'ENGINE': 'django.db.backends.mysql',
    'NAME': 'cmdbtest',
```

```
$ ll open-cmdb
total 32
-rw-r--r-- 1 liuyu staff 1948 Dec 25 09:45 README.md
drwxr-xr-x 6 liuyu staff 204 Jan 7 15:56 book
drwxr-xr-x 21 liuyu staff 714 Jan 7 16:16 cmdb
drwxr-xr-x 16 liuyu staff 544 Jan 7 16:16 echelon
-rw-r--r-- 1 liuyu staff 636 May 2 12:01 local_settings.pyc
drwxr-xr-x 1 liuyu staff 252 Dec 25 09:45 manage.py
drwxr-xr-x 14 liuyu staff 476 May 2 12:02 myproject
-rw-r--r-- 1 liuyu staff 80 Dec 25 09:45 requirements.txt
drwxr-xr-x 6 liuyu staff 204 Dec 25 09:45 static
drwxr-xr-x 12 liuyu staff 408 Jan 7 15:56 templates
drwxr-xr-x 8 liuyu staff 272 Dec 25 09:51 utils
```

图 2-8 Django 安装后的目录组织结构

```
'HOST': 'localhost',
'USER': 'root',
'PASSWORD': 'cmdbtest',
'PORT': '3306',
'OPTIONS': {'init_command': 'SET storage_engine=INNODB', 'charset': 'utf8', }
} }
```

(2) 设置 App，把我们新建的 CMDB App 加到末尾，代码如下：

```
INSTALLED_APPS = ('django.contrib.admin',
'django.contrib.auth',
'django.contrib.contenttypes',
'django.contrib.sessions',
'django.contrib.messages',
'django.contrib.staticfiles', 'cmdb',
)
```

(3) 设置静态文件存放目录：

```
STATICFILES_DIRS = ( os.path.join(BASE_DIR, 'static/'), )
```

(4) 设置模板文件存放目录：

```
TEMPLATE_DIRS = [ os.path.join(BASE_DIR, 'templates'), ]
```

(5) 设置登录 URL：

```
LOGIN_URL = '/cmdb/login/'
```

(6) 设置其他参数，可以根据自己的需求进行设置：

```
TEMPLATE_CONTEXT_PROCESSORS = ( 'django.core.context_processors.static', 'cmdb.
context_processors.menu', "django.contrib.auth.context_processors.auth",
'django.core.context_processors.request",
) CMDB_VERSION = '1.0' CMDB_NAME = u' 测试 CMDB' LOGIN_REDIRECT_URL = '/home/'
```

到此为止基础环境已经准备完毕，接下来需要设计数据库，并定义好视图。

## 5. 数据表设计

Django 遵循 MVC 设计，其中 M（模型）就是数据库模型，也就是 App 中的 models.py 文件的设置，Django 自带数据库的 ORM（Object Relational Mapping）架构，这使得我们不用再需要学习复杂的数据库操作，只需要通过定义 models 即可。下面是我设置的最简单的 CMDB 数据结构：

```
# 指定解析器为 Python
# !/usr/bin/env python
# 指定字符编码为 utf8
# encoding:utf8
# 从 django.db 中导入 models 模块
from django.db import models
# 导入 User 模块
```

```
from django.contrib.auth.models import User
# Create your models here.
# 定义一个 Server_Group 类, 从 models.Model 中继承, 这里就是所谓的数据表结构
class Server_Group(models.Model):
    # 定义主机组名称字段
    name = models.CharField(u'主机组', max_length=255, unique=True)
    # 关联的项目字段, 这里是关联一个外键
    project = models.ForeignKey("Project", verbose_name='项目名称')
    # 备注字段
    memo = models.CharField(u'备注', max_length=255, blank=True)
    # unicode 返回值
    def __unicode__(self):
        # 返回的格式
        return '%s-%s' % (self.project.name, self.name)

    # 定义 Meta 属性
    class Meta:
        # 数据库中的表名
        db_table = 'server_group'
        # 存储的时候需要确认组合键是唯一的
        unique_together = (("name", "project"),)

# 定义一个 IDC 类, 主要存储 IDC 信息, 数据表结构有 2 个字段
class IDC(models.Model):
    # 定义 IDC 的名称字段
    name = models.CharField(u'IDC 名称', max_length=255, unique=True)
    memo = models.CharField(u'备注', max_length=255, blank=True)

    def __unicode__(self):
        return self.name

    class Meta:
        db_table = 'idc'

# 定义一个 Project 类, 主要存储项目信息, 数据表结构有 2 个字段
class Project(models.Model):
    name = models.CharField(u'项目名称', max_length=255, unique=True)
    memo = models.CharField(u'备注', max_length=255, blank=True)

    def __unicode__(self):
        return self.name

    class Meta:
        db_table = 'project'

# 定义一个 Server_Role 类, 主要存储服务器角色信息, 数据表结构有 3 个字段
class Server_Role(models.Model):
    name = models.CharField(u'角色', max_length=255)
    # 关联 Server_Group, 也就是服务器组
    group = models.ForeignKey("Server_Group", verbose_name='项目组')
```

```

memo = models.CharField(u'备注', max_length=255, blank=True)

def __unicode__(self):
    return '%s-%s-%s' % (self.group.project.name, self.group.name, self.name)

class Meta:
    # 设置数据库表名
    db_table = 'server_role'
    # 存储的时候需要确认组合键是唯一的
    unique_together = (("name", "group"),)

# CMDB 核心数据表结构, 用来存储服务器系统信息
class Server_Device(models.Model):
    # 服务器状态选择, 具体的字段存储数据为 0 ~ 3 的 int 数字
    SERVER_STATUS = (
        (0, u'下线'),
        (1, u'在线'),
        (2, u'待上线'),
        (3, u'测试'),
    )
    # 定义一个名称字段, 若 blank 没有设置则默认为 False, 不能为空, 且 unique=True 必须唯一
    name = models.CharField(u'主机名称', max_length=100, unique=True)
    # 定义 SN 编号字段, blank=True, 可以为空
    sn = models.CharField(u'SN 号', max_length=200, blank=True)
    # 公网 IP 字段, 可以为空
    public_ip = models.CharField(u'外网 IP', max_length=200, blank=True)
    # 私网 IP 字段, 可以为空
    private_ip = models.CharField(u'内网 IP', max_length=200, blank=True)
    # 定义 MAC 地址字段
    mac = models.CharField(u'MAC 地址', max_length=200, blank=True)
    # 定义操作系统字段
    os = models.CharField(u'操作系统', max_length=200, blank=True)
    # 定义磁盘信息字段
    disk = models.CharField(u'磁盘', max_length=200, blank=True)
    # 定义内存信息字段
    mem = models.CharField(u'内存', max_length=200, blank=True)
    # 定义 CPU 信息字段
    cpu = models.CharField(u'CPU', max_length=200, blank=True)
    # 关联 IDC 信息
    idc = models.ForeignKey(IDC, max_length=255, blank=True, null=True, verbose_name='机房名称')
    # 定义一个多对多字段, 一台服务器可以对应多个角色
    role = models.ManyToManyField("Server_Role", verbose_name='角色', blank=True)
    # 机器状态, 默认都为在线状态
    status = models.SmallIntegerField(verbose_name='机器状态', choices=SERVER_STATUS, default=1)
    # 管理用户信息
    admin = models.ForeignKey('auth.User', verbose_name='管理员', null=True, blank=True)
    # 定义备注字段

```



```
memo = models.CharField(u'备注', max_length=200, blank=True)

def __unicode__(self):
    return self.name

class Meta:
    db_table = 'server_device'
```

初始化数据库,同时设置登录所需要的 username 和 password,命令如下:

```
$ python manage.py syncdb
Operations to perform:
Apply all migrations: admin, contenttypes, auth, sessions
Running migrations:
Applying contenttypes.0001_initial... OK
Applying auth.0001_initial... OK
Applying admin.0001_initial... OK
Applying sessions.0001_initial... OK
You have installed Django's auth system, and don't have any superusers defined.
Would you like to create one now? (yes/no): yes
# 这里输入用户名
Username (leave blank to use 'root'): admin
Email address:
# 这里输入密码
Password:
# 重复输入密码
Password (again):
Superuser created successfully.
```

在命令行登录数据库,并查看数据库信息,就能看到如图 2-9 所示的内容,说明数据库已创建成功。

## 6. 视图设置

上文中我们已经成功设置了 Django 的 M (Models, 模型),下面我们来设置 V (View, 视图),如下代码是一个登出页面和一个 home 页面的 View:

```
# encoding:utf8
# Create your views here.
# 导入需要使用的模块
from django.template import RequestContext
from django.shortcuts import render_to_response
from django.contrib.auth.decorators import login_required
from django.contrib.auth.views import logout_then_login
# 判断用户是否登录
@login_required
# 登出时的调用
def logout_view(request):
    return logout_then_login(request)
# 判断用户是否登录
```

```
[opencmdb] mysql> show tables;
+-----+
| Tables_in_opencmdb |
+-----+
| auth_group           |
| auth_group_permissions |
| auth_permission      |
| auth_user            |
| auth_user_groups     |
| auth_user_user_permissions |
| book_category        |
| chenlijun            |
| django_admin_log      |
| django_content_type   |
| django_migrations     |
| django_session        |
| echelon_changelogentry |
| idc                   |
| liuxia               |
| project              |
| provider             |
| server_device         |
| server_device_role    |
| server_group          |
| server_role           |
| testimport            |
| travel               |
| travel_line           |
| upload_file_message   |
+-----+
25 rows in set (0.00 sec)
```

图 2-9 查看数据库是否已创建

```
@login_required
# 登录后调用 home 页所展示的页面，template 为 home.html
def home(request):
    return render_to_response('home.html', locals(), context_instance=RequestCo
        ntext(request))
```

URL 设置（这里直接使用了 Django 自带的 login 函数，所以不需要自己写 login view）：

```
# 设置字符编码
# encoding:utf8
# 从 urls 中导入 patterns、include、url 模块
from django.conf.urls import patterns, include, url
# 从 contrib 中导入 admin 模块
from django.contrib import admin
# 从 http 中导入 HttpResponseRedirect 模块
from django.http import HttpResponseRedirect

# 设置前端访问的 URL 对应的后端视图
urlpatterns = patterns('',
    # url 什么参数都不带时，直接重定向到 login
    url(r'^$', lambda x: HttpResponseRedirect('/login/')),
    # 登出对应的视图为 cmdb.views.logout_view
    url(r'^logout/$', 'cmdb.views.logout_view', name='cmdb_logout'),
    # 登录对应的 view 为 django.contrib.auth.views.login，对应的
    # template 为 login.html
    url(r'^login/$', 'django.contrib.auth.views.login', {'template_
        name': 'login.html'},
        name='cmdb_login'),
    # home 页面，对应的 view 为 cmdb.views.home
    url(r'^home/$', 'cmdb.views.home', name='home'),
)
```

通过如上定义，现在就启动 Django 服务，登录后即可看到如图 2-10 所示的界面。代码已托管至 Github 网站 <https://github.com/oysterclub/open-cmdb>，有兴趣的朋友可以去复制下来查看、修改或使用。下面就来展示一下登录界面的效果图（注，前端框架为 bootstrap）。

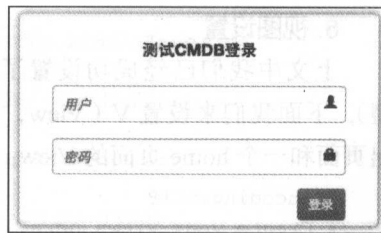


图 2-10 系统登录界面

利用 `python manage.py syncdb` 命令输入的用户名和密码登录。登录后的页面为 home 空白页（见图 2-11），具体如下（home 空白页主要是为了以后做导向流页面或数据图表展示页面，这里先留空）：

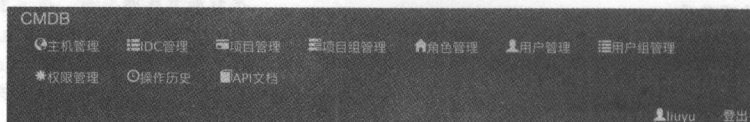


图 2-11 系统登录后的界面

## 7. 使用 Python 程序获取 Facts 数据

通过如上定义，我们已经完成了视图、数据表结构的定义。而数据的来源既可以通过添加，也可以通过 Facter 工具来获取。下面我们就来讲讲如何自动获取 Agent 机器的系统数据（如果想要充分了解 Facter 工具，可以参考阅读《Puppet 实战》的第9章“Facter 介绍”）。Facter 工具会在 Puppet Agent 与 Puppet Master 通信的时候把获取到的 Agent 主机系统信息和自己定义的 Facts 信息汇报给 Puppet Master，生成一个 hostname.yaml 格式的文件，文件存放在 /var/lib/puppet/yaml/facts 目录下，文件的格式如图 2-12 所示，其中的 values 数据：domain、ipaddress、macaddress 等正是构建 CMDB 所需要的系统数据，因此我们可以通过一个 Python 程序来处理这些数据，并录入 MySQL 中，最终通过 Django 来实现前端展示。因此一个最简单的 CMDB 系统就构建完成了。

```
#2#root@ ~ -j$ more /var/lib/puppet/yaml/facts/toml.yaml
--- !ruby/object:Puppet::Node::Facts
  expiration: 2015-12-29 17:44:03.149511 +08:00
  name: toml
  values:
    memorysize_mb: "3791.53"
    netmask: "255.255.255.0"
    fqdn: toml
    swapsize: "8.00 GB"
    kernelversion: "3.10.0"
    blockdevice_sda_model: "QEMU HARDDISK"
    ip6tables_version: "1.4.21"
    processor2: "QEMU Virtual CPU version (cpu64-rhel6)"
    macaddress_eth1: "52:54:00:ce:65:a8"
    uniqueid: "00000000"
    blockdevices: "sda,sr0"
    processors: "physicalcount4models0QEMU Virtual CPU version (cpu64-rhel6)QEMU Virtual
    U version (cpu64-rhel6)QEMU Virtual CPU version (cpu64-rhel6)count4"
    selinux: "false"
    is_virtual: "true"
    productname: KVM
    memoryfree: "3.36 GB"
    virtual: kvm
    network_lo: "127.0.0.0"
```

图 2-12 Facter 上报至 Puppet Master 后的 yaml 部分信息

我们先来具体看一下 `facter_message.py` 程序（Python 程序处理 Facter 数据），完整代码如下：

```
#!/usr/bin/env python
# encoding: utf8
__authors__ = ['liuyu', 'chenlijun']
__version__ = 1.0
__date__ = '2015-09-06 14:58:23'
__licence__ = 'GPL licence'

# 导入模块
import yaml
import os
# IPy 主要用来判断 IP 类型，IPy.IP('ip').iptype()
import IPy

# yaml 文件目录
```

```
yaml_dir = '/var/lib/puppet/yaml/facts'
```

```
# 结果集, 结果集的格式 {'cmdb_agent': {}}
```

```
all_host_facter_message = {}
```

```
# 结果列表
```

```
result_list = ['name',
```

```
    'SN',
```

```
    'public_ip',
```

```
    'private_ip',
```

```
    'mac',
```

```
    'os',
```

```
    'disk',
```

```
    'mem',
```

```
    'cpu',
```

```
    'idc',
```

```
    'role',
```

```
    'status',
```

```
    'admin',
```

```
    'memo']
```

```
# db 对应的 Facter 字段, 需要获取其他的字段时可以一一对应
```

```
list_field = {'name': 'fqdn',
```

```
    'public_ip': 'ipaddress__interfaces',
```

```
    'private_ip': 'ipaddress__interfaces',
```

```
    'mac': 'macaddress__interfaces',
```

```
    'os': ['operatingsystem', 'operatingsystemrelease', 'hardwaremodel'],
```

```
    'disk': 'blockdevice__blockdevices',
```

```
    'mem': 'memorysize',
```

```
    'cpu': ['processorcount', 'processor0']}
```

```
# ruby objectobjectconstruct
```

```
def construct_ruby_object(loader, suffix, node):
```

```
    return loader.construct_yaml_map(node)
```

```
def construct_ruby_sym(loader, node):
```

```
    return loader.construct_yaml_str(node)
```

```
# 读取数据
```

```
def yaml_file_handle(filename):
```

```
    stream = open(filename)
```

```
    mydata = yaml.load(stream)
```

```
    return mydata
```

```
# 获取 IP 的类型
```

```
def get_ip_type(ip):
```

```
    try:
```

```
        return IPy.IP(ip).iptype().lower()
```

```
    except Exception, e:
```

```
        print e
```

```

# 处理单个 Agent 的数据
def handle_factor_message(data):
    # 定义一个结果字典, 字段和 db 一样, 处理完的结果和 db 中的一样
    result_dict = {}
    # 对结果进行处理
    for db_field in result_list:
        # 定义一个字段结果字符串
        value = ''
        # result_list 中的字段是否存在于我们需要的 Factor 取值列表中, 如果存在
        if db_field in list_field:
            factor_field = list_field[db_field]
            # 先判断 factor_field 的类型, 然后进行处理
            if type(factor_field) == type([]):
                for tag in factor_field:
                    if data.get(tag):
                        value += data[tag] + ' '
            else:
                # 由于 disk、IP 等需要进一步处理, 所以用了一个 __ 来分隔, 然后再进行处理
                field_tmp = factor_field.split("__")
                if len(field_tmp) == 2:
                    if db_field == 'disk':
                        for tag in data[field_tmp[1]].split(","):
                            # 对磁盘进行处理, 由于磁盘的字段为 blockdevice_type_size,
                            # 所以需要单独进行处理
                            f = field_tmp[0] + '_' + tag + '_' + 'size'
                            if data.get(f):
                                # 去除 sr0 tag 的字段
                                if tag != 'sr0':
                                    # 结果字符串
                                    value += tag + ':' + str(int(data[f]) / 1024 /
                                                                1024 / 1024) + 'G' + ' '
                    # 对外网 IP 进行处理
                    elif db_field == 'public_ip':
                        for tag in data[field_tmp[1]].split(","):
                            f = field_tmp[0] + '_' + tag
                            if data.get(f):
                                # 去除 lo tag 的字段
                                if tag != 'lo':
                                    if get_ip_type(data[f]) == 'public':
                                        # 结果字符串
                                        value += data[f] + ' '
                    # 对内外 IP 进行处理
                    elif db_field == 'private_ip':
                        for tag in data[field_tmp[1]].split(","):
                            f = field_tmp[0] + '_' + tag
                            if data.get(f):
                                # 去除 lo tag 的字段
                                if tag != 'lo':
                                    if get_ip_type(data[f]) == 'private':
                                        # 结果字符串

```

```

        value += data[f] + ' '
    else:
        # 其他的字段直接就处理了
        for tag in data[field_tmp[1]].split(","):
            f = field_tmp[0] + '_' + tag
            if data.get(f):
                # 去除 lo tag 的字段
                if tag != 'lo':
                    # 结果字符串
                    value += tag + ':' + data[f] + ' '
            else:
                if data.get(facter_field):
                    # 结果字符串
                    value = data[facter_field]
                # 将结果添加到 result 列表中
                result_dict[db_field] = value.strip()
            # 如果不存在
        else:
            result_dict[db_field] = ''
    # 返回结果字典
    return result_dict

# 定义获取 facter 的函数
def get_all_host_facter_message():
    # 由于 Puppet 的 yaml 文件是 Ruby 格式的，因此需要进行转换
    yaml.add_multi_constructor(u"!ruby/object:", construct_ruby_object)
    yaml.add_constructor(u"!ruby/sym", construct_ruby_sym)
    # 获取所有有 Facter 信息的主机文件名称
    for dirpath, dirnames, filenames in os.walk(yaml_dir):
        # 只需要处理 yaml 目录下以 yaml 结尾的文件
        if dirpath == yaml_dir:
            for file in filenames:
                file_name, file_ext = os.path.splitext(file)
                if file_ext == '.yaml':
                    host_yaml_path = yaml_dir + '/' + file
                    # 得到 yaml 文件的内容，字典形式
                    host_yaml_result_dict = yaml_file_handle(host_yaml_path)
                    # 对单个 Agent 的数据进行处理
                    if host_yaml_result_dict:
                        # 由于有 key 为 facts，所以可以直接查找 facts key 的值
                        if host_yaml_result_dict.has_key('facts'):
                            data_dict = host_yaml_result_dict['facts']['values']
                            # 没有的就直接取
                        else:
                            data_dict = host_yaml_result_dict['values']

                    # 现在就可以对 data 进行处理，获取我们所需要的数据了
                    result_dict = handle_facter_message(data_dict)
                    all_host_facter_message[file_name] = result_dict
    # 返回我们最终的数据结果集

```



```
return all_host_factor_message
```

以上程序可以过滤 Factor 中我们想要得到的 Agent 数据，运行 factor\_message.py 程序，结果输出如下：

```
$ python factor_message.py
{'puppetclient.domain.com':
{'status': '',
'name': 'puppetclient.domain.com',
'mem': '1.83 GB',
'memo': '',
'idc': '',
'public_ip': '',
'admin': '',
'mac': 'eth0:00:1A:4A:25:E2:12 eth1:00:1A:4A:25:E2:13',
'role': '',
'private_ip': '10.20.122.111',
'disk': 'vda:20G vdb:30G',
'os': 'CentOS 6.5 x86_64',
'cpu': '2 Intel Core 2 Duo P9xxx (Penryn Class Core 2)',
'SN': ''}
}
```

到这里，我们能够看到 factor\_message.py 得到的数据字段和 models.py 中数据结构的字段正好一样，下一步我们就可以直接将 factor\_message.py 的数据导入到数据库中了，具体程序如下：

```
# 检测用户是否登录
@login_required
# 定义一个 views，用来处理导入信息
def import_data(request, model):
    # 导入计数器
    import_num = 0
    # 查看 model 是否存在于定义的模板中
    if model in BASE_ADMIN:
        # 获取 tag 名称
        tag_name = BASE_ADMIN[model]['name']
        # 获取 model 名称
        model_name = BASE_ADMIN[model]['model']
        # 这里只处理 server_device 的导入信息
        if model == 'server_device':
            server_device_data = get_all_host_factor_message()
            # 进行数据入库处理
            for hostname, factor_message in server_device_data.items():
                # 主机名处理，判断 factor_message 中 name key 是否有值，
                if factor_message['name']:
                    # 如果有值，name 就使用该值
                    name = factor_message['name']
                # 如果没有这个值
```

```

else:
    # 就使用 hostname
    name = hostname
    # 对于 IDC 信息、User 信息、项目角色信息的处理都需要自己去写 Facter 插件，不
    # 写的都为空，然后进行处理
    # IDC 关联处理，如果 facter_message 中的 idc key 有值
    if facter_message['idc']:
        # idc_name 就为该值
        idc_name = facter_message['idc']
        # 同时处理该 IDC 信息是否存在于 IDC 表中，如果有则取出 ID
        if IDC.objects.filter(name=idc_name):
            idc_id = IDC.objects.get(name=idc_name).id
            # 如果没有，则进行保存，然后取出 ID
        else:
            idc_sql = IDC(name=idc_name)
            try:
                idc_sql.save()
            # 取出 ID
            idc_id = IDC.objects.get(name=idc_name).id
            except Exception, e:
                return e
    # 如果 idc key 没有值，则为 None
    else:
        idc_id = None
    # 管理员信息关联处理，如果用户存在则关联，不存在则跳过
    if facter_message['admin']:
        admin_name = facter_message['admin']
        # 如果用户存在 User 表中则取 ID，若没有则为空
        if User.objects.filter(username=admin_name):
            user_id = User.objects.get(username=admin_name).id
        else:
            user_id = None
    # 没有就为空
    else:
        user_id = None
    # 这里还有一个角色多对多关系的处理，由于这里没有定义机器角色，因此此处不处理
    # 角色信息
    # 判断主机是否存在于 server_device 表中，如果不存在则添加
    if not model_name.objects.filter(name=name):
        import_sql = model_name(name=name,
                                sn=facter_message['sn'],
                                public_ip=facter_message['public_ip'],
                                private_ip=facter_message['private_ip'],
                                mac=facter_message['mac'],
                                idc=idc_id,
                                os=facter_message['os'],
                                disk=facter_message['disk'],
                                mem=facter_message['mem'],
                                cpu=facter_message['cpu'],
                                admin=user_id,

```

```

memo=facter_message['memo'],
    )
    try:
        # 保存
        import_sql.save()
    except Exception, e:
        return e
# 如果有了, 则查询数据, 若信息不对则更新
elif not model_name.objects.filter(name=name,
    sn=facter_message['sn'],
    public_ip=facter_message['public_ip'],
    private_ip=facter_message['private_ip'],
    mac=facter_message['mac'],
    os=facter_message['os'],
    disk=facter_message['disk'],
    mem=facter_message['mem'],
    cpu=facter_message['cpu'],
    memo=facter_message['memo']):
    try:
        # 更新数据库
        model_name.objects.filter(name=name).update(sn=
            facter_message['sn'],
            public_ip=facter_message['public_ip'],
            private_ip=facter_message['private_ip'],
            mac=facter_message['mac'],
            os=facter_message['os'],
            disk=facter_message['disk'],
            mem=facter_message['mem'],
            cpu=facter_message['cpu'],
            memo=facter_message['memo'],
        )
    except Exception, e:
        return e
# 如果有了, 且信息 ok, 则跳过
else:
    continue
return HttpResponseRedirect('/cmdb/%s/show/' % model)

return render_to_response('all_data_show.html', locals(), context_instance=
RequestContext(request))

```

重新登录 CMDB 之后的页面, 有一个导入主机的按钮, 点击导入主机按钮, 就可以自动导入通过 Factor 获取的 Agent 主机信息了, 如图 2-13 所示。

我们还可以通过添加的方式来维护 CMDB 的内容, 到目前为止我们已经完成了使用 Python 和 Puppet 来构建一个小型的、简单的 CMDB 系统。

CMDB 是需要我们定期进行维护和更新的, 因此它还需要提供历史查看、API 等更实用的功能, 为此在 2.6 节中我们将介绍一下 Django 提供的几个好用的功能模块。

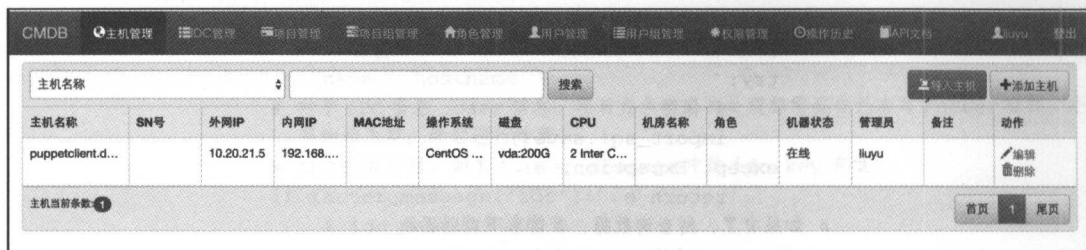


图 2-13 信息导入界面

## 2.6 高级进阶

### 2.6.1 历史查询功能

操作历史，应该是管理系统必备的功能之一。Django-echelon 就是一个很好的功能模块，该功能模块可以单独使用于任何的 Django 项目中，非常方便，接下来我们讲解如何将该功能添加到 CMDB 系统中，设置步骤具体如下。

(1) 修改 settings.py 设置。

①添加 echelon APP:

```
INSTALLED_APPS = (
    '.....',
    'echelon',
)
```

②添加 echelon 中间件:

```
MIDDLEWARE_CLASSES = (
    '.....',
    'echelon.middleware.EchelonMiddleware',
)
```

(2) 修改 urls.py 设置:

```
# encoding:utf8
from django.conf.urls import patterns, include, url
urlpatterns = patterns('',
    # .....
    # 操作历史
    url(r'^cmdb/changelog/', include('echelon.urls')),
)
```

(3) 将 echelon 代码复制到 Django 项目的主目录中，将 html 代码复制到 templates 目录中。

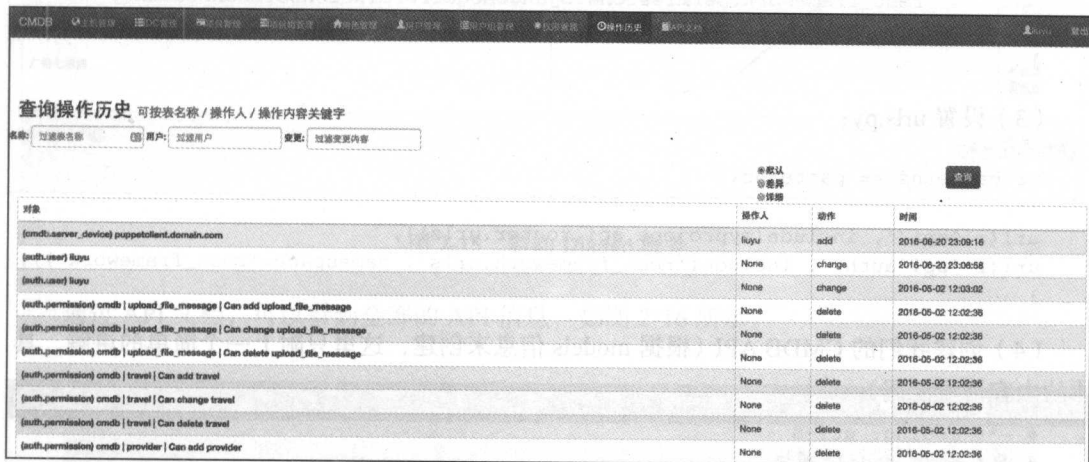
(4) 刷新 DB，创建数据表:

```
$ python manage.py syncdb
```

(5) 把操作历史的 URL 信息添加到前端页面的导航栏中就可以了(配置 cmdb/cmdb\_menu.py 程序), 具体操作如下:

```
CMDB_TOP_MENU = [
    '.....',
    # 导航名称、URL、图标、子导航信息
    ['u' 操作历史 ', '/cmdb/changelog/', 'time', []],
]
```

(6) 启动 Django 服务, 打开前端页面, 然后点击操作历史, 就可以看到用户的操作信息了(见图 2-14), 具体如下。



对象	操作人	动作	时间
(cmdb.server.device) puppetclient.domain.com	liuyu	add	2016-06-20 23:09:16
(auth.user) liyu	None	change	2016-06-20 23:06:58
(auth.user) liyu	None	change	2016-06-02 12:03:02
(auth.permission) cmdb   upload_file_message   Can add upload_file_message	None	delete	2016-06-02 12:02:36
(auth.permission) cmdb   upload_file_message   Can change upload_file_message	None	delete	2016-06-02 12:02:36
(auth.permission) cmdb   upload_file_message   Can delete upload_file_message	None	delete	2016-06-02 12:02:36
(auth.permission) cmdb   travel   Can add travel	None	delete	2016-06-02 12:02:36
(auth.permission) cmdb   travel   Can change travel	None	delete	2016-06-02 12:02:36
(auth.permission) cmdb   travel   Can delete travel	None	delete	2016-06-02 12:02:36
(auth.permission) cmdb   provider   Can add provider	None	delete	2016-06-02 12:02:36

图 2-14 用户的操作信息

以上代码都可以在 open-cmdb 库上找到源代码。

## 2.6.2 API 功能

CMDB 作为一个数据源中心, 很多运维工具都会调用 CMDB 数据进行使用, 因此 API 接口就非常有必要了, 由于各个系统的需求不一样, 从头到尾开发一套适用于各个系统的 CMDB API 也比较困难, 那么有没有什么简单的方法呢? 答案肯定是有, 这里我推荐 Django API 利器 Django REST framework。

Django REST framework 是一个非常强大、灵活的 API 构建工具, 它能很容易、很快速地帮我们构建 Web API。下面来讲解构建的过程, 具体步骤如下。

(1) 安装 Django REST framework。

主要安装 3 个模块: djangorestframework、markdown、django-filter。具体代码如下:

```

pip install djangorestframework
pip install markdown
pip install django-filter

```

## (2) 配置 settings.py:

```

INSTALLED_APPS = (
    '.....',
    'rest_framework',
)
REST_FRAMEWORK = {
    # Use Django's standard `django.contrib.auth` permissions,
    # or allow read-only access for unauthenticated users.
    'DEFAULT_PERMISSION_CLASSES': [
        'rest_framework.permissions.DjangoModelPermissionsOrAnonReadOnly'
    ]
}

```

## (3) 设置 urls.py:

```

urlpatterns += patterns(
    '',
    url(r'^api/', include(myproject.api.router.urls)),
    url(r'^api-auth/', include('rest_framework.urls', namespace='rest_framework')),
)

```

(4) 创建我们的 CMDB API (根据 models 信息来创建, 这里只附上一个简单的讲解, 具体的内容请看源码):

```

# -*- coding: utf-8 -*-
# 导入 cmdb.models 模块
import cmdb.models
# 从 rest_framework 中导入模块
from rest_framework import routers, serializers, viewsets

# 给需要生成 API 的 model 定义一个数据序列
# Serializers define the API representation.
class IDCSerializer(serializers.HyperlinkedModelSerializer):
    class Meta:
        # 使用的 model 名称
        model = cmdb.models.IDC
        # 字段序列
        fields = ('url', 'name', 'memo')

# 定义视图
class IDCViewSet(viewsets.ModelViewSet):
    # 查询所有数据, 这个可以根据自己的需要来展示几个
    queryset = cmdb.models.IDC.objects.all()
    # 序列化信息
    serializer_class = IDCSerializer

```



```
# 配置路由注册, 自动生成 API URL
# Routers provide an easy way of automatically determining the URL conf.
router = routers.DefaultRouter()
router.register(r'idcs', IDCViewSet)
```

### (5) 配置导航栏信息:

```
cmdb_menu.py CMDB_TOP_MENU = [ [u'API 文档 ', '/api/', 'book', []], ]
```

### (6) 启动 Django 服务, 展示如下 (图 2-15)。

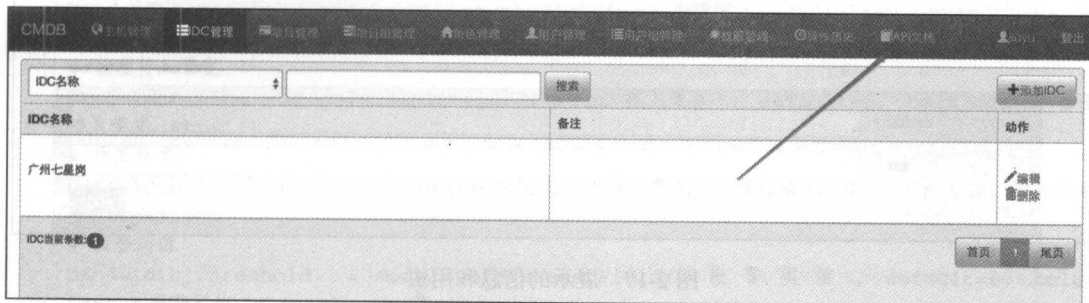


图 2-15 启动 Django 服务

通过 API 文档, 可以看到全部的 API 信息, 如图 2-16 所示。

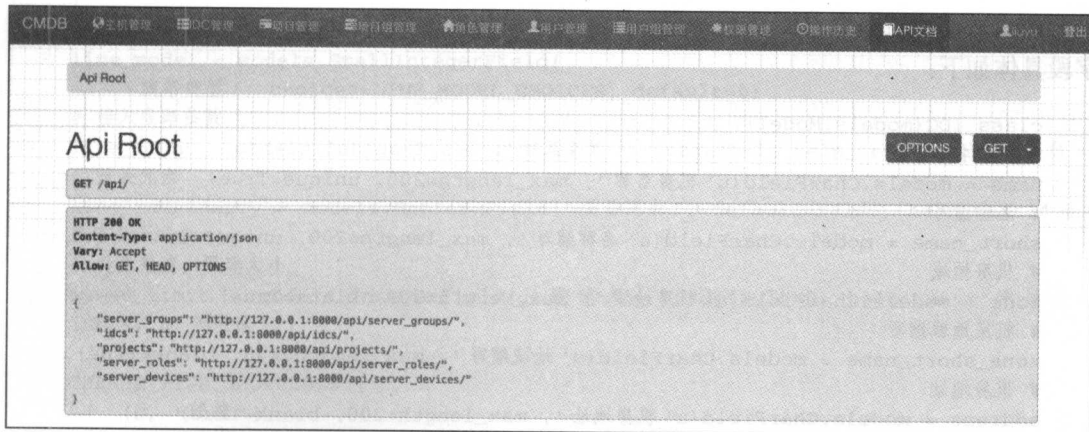


图 2-16 全部的 API 信息

点击 IDC 的具体 URL, 就能看到展示的信息和用法, 如图 2-17 所示。

到这里 CMDB API 就完成了最基本的功能, 后续还需要读者自己去调整页面和授权等。

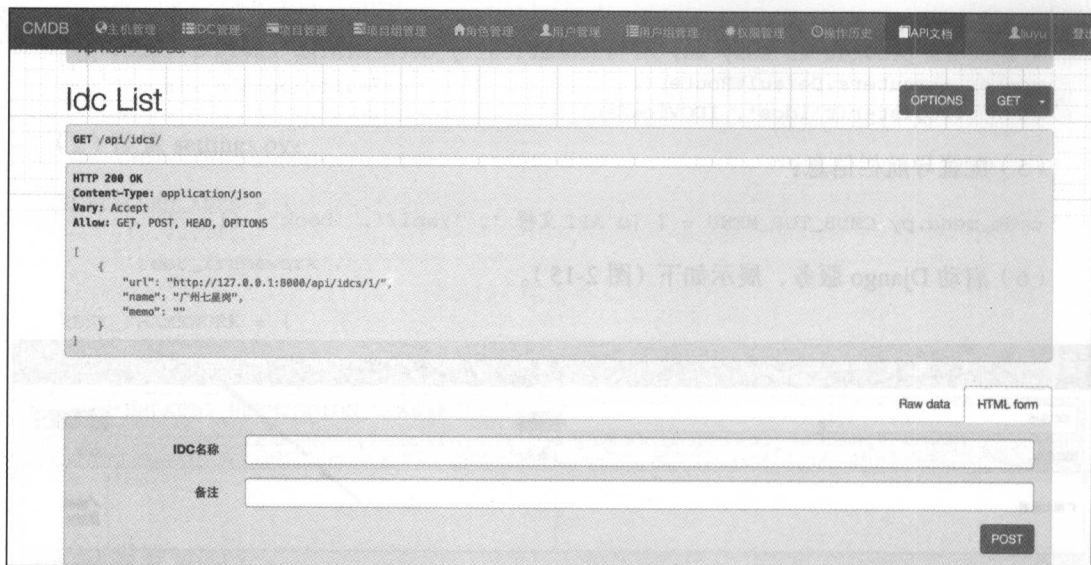


图 2-17 展示的信息和用法

### 2.6.3 数据表结构

对于数据表结构，前面的定义是比较简单的，有兴趣的读者可以根据自己的实际需求去增加，这里仅列举一个简单的优化例子——IDC 表优化。

IDC 表前面就一个 IDC 名称字段，这里将根据作者所在公司的需求来做优化，优化后的字段具体如下：

```
class IDC(models.Model):
    # 机房名称字段
    name = models.CharField(u'机房名称', max_length=200, unique=True)
    # 名称缩写
    short_name = models.CharField(u'名称缩写', max_length=200, unique=True)
    # 机房地域
    zone = models.CharField(u'机房地域', max_length=200, blank=True)
    # 机房地域缩写
    zone_short_name = models.CharField(u'地域缩写', max_length=200, blank=True)
    # 机房地址
    address = models.CharField(u'机房地址', max_length=200, blank=True)
    # 机柜U数
    unit_size = models.PositiveIntegerField(u'机柜U数', default=42, help_text="机房每个机柜拥有 Unit 数目")
    # 机柜电数
    power_size = models.PositiveIntegerField(u'机柜电数', default=12, help_text="机房每个机柜拥有电力 (An) 数目")

INTERFACE_TYPE_CHOICES = (
```

```

        (0, ' 光纤 '),
        (1, ' 普通 '),
        (2, ' 其他 '),
    )
    # 网络接入方式是光纤、普通还是其他
    interface_type = models.SmallIntegerField(
        ' 网络接入 ', choices=INTERFACE_TYPE_CHOICES, default=0)
    # 机房所有设备的总带宽
    bandwidth_equipment = models.BigIntegerField(u' 设备带宽 ', default=0, help_text="
    设备带宽 (Mbps)")
    # 机房可用带宽
    bandwidth_usable = models.BigIntegerField(u' 可用带宽 ', default=0, help_text="
    使用最大带宽 (Mbps)")
    # 物理接入带宽
    bandwidth_line = models.BigIntegerField(u' 接入带宽 ', default=0, help_text=" 物理
    接入带宽 (Mbps)")
    # 保底带宽
    bandwidth_min = models.BigIntegerField(u' 保底带宽 ', default=0, help_text=" 保底带
    宽 (Mbps)")
    # 报警阈值
    bandwidth_threshold = models.BigIntegerField(u' 报 警 阈 值 ', default=0, help_
    text=" 报警阈值 (Mbps)")

    LINE_MODEL_CHOICES = (
        (0, ' 单线 '),
        (1, ' 双线 '),
        (2, ' BGP '),
    )
    # 机房线路类型是单线、双线还是 BGP
    line_model = models.SmallIntegerField(
        ' 线路类型 ', choices=LINE_MODEL_CHOICES, default=0)
    # 接入线路条数
    line_count = models.IntegerField(u' 线路条数 ', default=1, help_text=" 接入线路条数 ")
    # 是否冗余
    line_redundancy = models.BooleanField(" 是否冗余 ", default=False, help_text=" 接入
    线路是否冗余 ")
    # 是否限速, 限速大小
    speed_limit = models.IntegerField(u' 限速大小 ', default=0, help_text=" 限速大小
    (Mbps, 0: 不限速 )")

    CHARGE_TYPE_CHOICES = (
        (0, ' 峰值 '),
        (1, ' 95 值 '),
        (2, ' 保底 '),
        (3, ' 计时 '),
        (4, ' 包月 '),
        (5, ' 包年 '),
    )
    # 计费模式
    charge_type = models.SmallIntegerField(

```

```

    '计费模式', choices=CHARGE_TYPE_CHOICES, default=0)
# 是否直连 ISP
isp_direct_link = models.BooleanField("是否直连 ISP", default=False)
# 所属运营商, 这里需要再建立一个 ISP 的类
isp = models.ForeignKey("ISP", verbose_name="运营商")
# 联系人
contact = models.ForeignKey(UserProfile, related_name="idc_contact", verbose_name="联系人", help_text="IDC 机房联系人")
# IDC 经理
manager = models.ForeignKey(UserProfile, related_name="idc_manager", verbose_name="经理", help_text="IDC 经理")
# 值班人
duty_officer = models.ForeignKey(UserProfile, related_name="duty_officer", verbose_name="值班人", help_text="IDC 机房值班联系人")
# IDC 报障邮箱
support_email = models.EmailField(u'报障邮箱', help_text="IDC 报障邮箱")
# 公司联系人
company_contact = models.ForeignKey(UserProfile, related_name="xsj_contact", verbose_name="公司联系人", help_text="公司联系人")
# 开通时间
start_time = models.DateTimeField(u'开通时间',
                                   auto_now_add=True)

# 到期时间
end_time = models.DateTimeField(u'到期时间',
                                default=datetime.datetime(2140, 1, 1, 0, 0, 1))

memo = models.CharField(u'备注', max_length=200, blank=True)

def __unicode__(self):
    return self.name

class Meta:
    db_table = 'idc'

```

现在这个表结构就是一个复杂的数据表结构, 通过修改 `base_admin.py` 就可以直接展示了。

## 2.6.4 用户管理功能

Django 有一套自己的用户管理、用户组管理和权限管理, 我们在初始化 Django 项目的时候, Django 会默认创建 User、Permission、Group 这 3 个表及对应的关联关系表。

本章所介绍的这套简单的 CMDB 系统, 已经将展示、添加、编辑和删除都模块化了。通过模板构建用户管理也是一件很轻松的事情, 只需要把 Django 自带的用户信息提取出来套用模板即可, 具体步骤如下。

(1) 在 `cmdb/base_admin.py` 中设置。

导入 User 模块:

```

from django.contrib.auth.models import User

'user': {
# model 名称
'model': User,
# form 表单
'form': User_CheckFrom,
# 名称 'name': u' 用户管理 ',
# 是否可以导入信息
'import': '',
# table 展示字段
'list_display': ['username',
'password', 'email', 'is_superuser', 'is_active', 'is_staff',
'groups',
'user_permissions'],
# 编辑只读字段
'readonly': ['username'],
# 动作
'action_list': [(u' 编辑 ', 'pencil', '/cmdb/user/modify/'),]
},

```

(2) 把用户管理的 URL 信息添加到前端页面的导航栏中。(配置 cmdb/cmdb\_menu.py 程序), 具体操作如下:

```

CMDB_TOP_MENU = [
'.....',
# 导航名称、URL、图标、子导航信息
[u' 用户管理 ', '/cmdb/user/show/', 'user', []],
]

```

(3) 启动 Django 服务, 打开前端页面, 点击用户管理, 信息如图 2-18 所示。



图 2-18 用户组管理信息

(4) 到这里一个简单的用户管理功能就实现了, 如果读者需要更详细的用户信息, 可以去创建一个 UserProfile 表, 然后关联到 Django 自带的 User 信息中, 限于篇幅, 本章就不逐一详细介绍了。

## 2.6.5 用户组管理功能

用户组管理和用户管理的原理是相通的, 添加方式也是一样的, 这里只是简述讲解一下

添加的步骤。

(1) 在 `cmdb/base_admin.py` 中设置：

```
'group': {
# model 名称
'model': Group,
# form 表单
'form': Group_CheckFrom,
# 名称
'name': u' 用户组管理 ',
# 是否可以导入信息
'import': '',
# table 展示字段
'list_display': ['name',
'permissions'],
# 编辑只读字段
'readonly': ['name', ],
# 动作
'action_list': [(u' 编辑 ', 'pencil', '/cmdb/group/modify/'),]
},
```

(2) 把用户组管理的 URL 信息添加到前端页面的导航栏中就可以了（配置 `cmdb/cmdb_menu.py` 程序），具体操作如下：

```
CMDB_TOP_MENU = [
'.....',
# 导航名称、URL、图标、子导航信息
[u' 用户组管理 ', '/cmdb/group/show/', 'list', []],
]
```

(3) 启动 Django 服务，打开前端页面，点击用户组管理（默认为空），信息如图 2-19 所示。

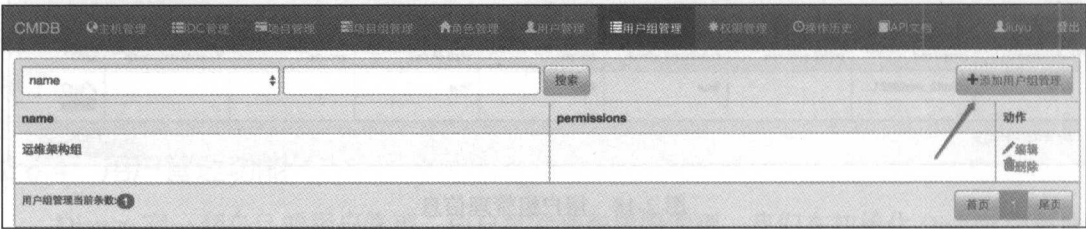


图 2-19 添加用户组管理

(4) 到这里组管理功能也实现成功了，现在我们就可以创建自己想要的组信息，把相应的用户添加到组中了。

到此为止，我们已经基于 CMDB 完善了用户、用户组、操作历史、开放 API 几大实用功能。通过这些构建，可以将 CMDB 和自动化运维平台的其他系统逐一打通，构建属于自己的平台。

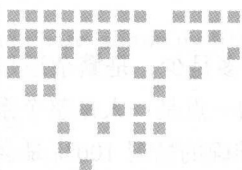


## 2.7 小结

无论 IT 系统多么庞大或微小，也无论提供的服务是多还是少，是繁杂还是单一；作为实际管理和掌控这套系统和服务的读者来说，必须要明确的一点是：人是整个系统中最薄弱和不可预测的一环。如果由人工来进行维护，那么数据不准确的情况 100% 是会出现的。而通过程序自动采集数据，进行规则入库这种自动化操作，则使数据达到 100% 的准确率成为了现实。

本文所展现的快速构建方法，仅仅是整套优化、完善动作的持续进行步骤中的一个环节，但此方法不但可以实现 CMDB 的快速构建，还对整个运维体系的建立、自动化平台架构的完善，也有非常重要的参考价值和借鉴意义。

最后希望读者能通过对本文的阅读，不仅能够构建出属于自己的、定制化的 CMDB 平台，还能通过更深入的思考，为完善运维工作、完美自动化平台增加更多更深刻的理解认知。



## Chapter 3 第3章

# 数据中心搬迁中的 x86 自动化运维

### 作者简介

吴传玉，具有 10 年以上 x86 服务器平台系统管理经验，熟悉 Windows 及 Redhat Linux 系列运维，自 2008 年接触 VMware 虚拟化产品，获得 RHCT、VMware VCP 和 HCNA 认证。善于使用各类脚本工具编制日常运维的脚本。目前就职于某大型金融企业基础设施运维部门，负责容灾及研发测试环境 x86 整体运维工作。

本章主要介绍在大型数据中心搬迁的过程中，如何利用自行编制的各类脚本，低成本、高效率又准确地完成大量节点的逻辑搬迁工作。

## 3.1 数据中心搬迁准备

### 3.1.1 数据中心搬迁介绍

世间万物都存在着自身的生命周期，大到宇宙恒星的诞生与消亡，小到细胞从一次分裂完成开始到下一次分裂结束所经历的全过程，都离不开生命周期。

作为承载着公司核心竞争、运维能力的数据中心，也无法回避生命周期的问题。当数据中心的空间、电力等基础硬件无法支持公司业务需求的高速扩展时，公司的决策层势必会考虑采用更合理的方案，以满足日益增长的业务需求。第一种是在保持现有数据资源不变的情况下，建立第二个数据中心（非容灾中心）；另一种是新建一个条件更优越的数据中心以替代旧的数据中心。

采用第一种方式，可以平滑地进行扩容，但从长远来看双倍的运维成本会加重公司的负

担。如果采用第二种方式,虽然在搬迁过程中对基础层、应用层运维人员的要求相对较高,但从长远来看可为公司节省可观的运维开支,主要的成本仅在于一次性搬迁的方案所承担的开支。

大型金融企业中,基础运维人员相对于应用运维、开发人员、测试人员来说是属于更底层、更核心的角色。从整体运维的宏观结构来看,基础运维位于一个漏斗的底部出口位置,支撑着众多应用业务系统的基础需求。总结起来就是底层硬件架构多元化、需求种类多、运维量大,如果涉及搬迁则工作量更大更繁琐。

因我主要负责的是 x86 平台的非生产环境日常运维工作,且正参与数据中心的搬迁项目。故本章将从系统层基础运维人员的角度出发,介绍在大型数据中心搬迁的过程中,如何自行编制脚本,从而降低成本、高效准确地完成大量节点的逻辑搬迁工作<sup>①</sup>。

首先介绍一下物理搬迁与逻辑搬迁的对比。

数据中心的搬迁一般分为两类:物理搬迁和逻辑搬迁。

如果物理设备及其上的逻辑节点数量少且搬迁距离较短,对可用性的要求不高,能够进行短期的停机,那么这种小型数据中心,一般较适用于同城数据中心的物理搬迁。

如果涉及的节点数量多且搬迁距离较长,对可用性要求较高,无法满足短期停机的要求,则需要考虑逻辑搬迁。

### 3.1.2 搬迁环境介绍

此次我所参与的搬迁存在诸多制约因素,如硬件种类繁多、应用关联性大、搬迁周期短、搬迁成本缩减等。基于以上现状分析,最终确定按应用环境,逐批次实施逻辑搬迁。

搬迁批次的定义如表 3-1 所示。

表 3-1 搬迁批次定义

批次	搬迁环境	资源准备及搬迁思路
1	容灾环境	<p>①新购计算资源放置在目标数据中心,预先完成虚拟化环境的部署</p> <p>②新购存储放置在源数据中心,通过存储复制技术完成灾备逻辑节点的复制。将完成复制的新存储物理搬迁至目标数据中心,释放之上的逻辑节点至新购计算资源,完成后续配置</p> <p>注意:由于节点量大,通过广域网链路进行存储复制对带宽的要求相当高,故采用局域网内完成存储复制的工作</p>
2	研发环境	<p>①利用批次 1 中已完成逻辑迁移的存储资源,通过存储复制技术完成研发测试环境逻辑节点的复制。将批次 1 留下的计算资源与完成研发测试环境复制的存储资源物理搬迁至目标数据中心,释放之上的逻辑节点,完成后续配置</p> <p>②对批次 2 中研发测试环境已完成逻辑搬迁的计算与存储资源进行物理搬迁,补充至目标数据中心</p> <p>注意:循环利用上批次释放的物理资源进行后续批次搬迁,可以有效地控制硬件成本</p>

① 本文介绍的脚本对于逻辑搬迁和物理搬迁均适用。

### 3.1.3 搬迁前的准备工作

搬迁过程不仅仅是物理设备位置的改变，还需要对虚拟化层及各个节点内的参数进行调整。由于金融企业的环境特殊，公司安全部门禁止自行配置自动化运维管理工具，因此只能利用现有的技术做文章了。

在此次迁移过程中，我利用了平时运维常用的一些系统自带的脚本工具。

这些工具分为三类，其中有用于虚拟化层的工具 vCLI、PowerCLI，有用于 Linux 环境的 Bash，还有用于 Windows 环境的批处理、WMIC 和注册表。

❑ vCLI: vSphere CLI，命令行管理接口。

❑ PowerCLI: VMware vSphere PowerCLI，是一款功能强大的命令行工具，可自动执行 vSphere 的各方面管理，包括主机、网络、存储、虚拟机、客户操作系统等。

❑ Bash: Bourne-Again Shell，是一个为 GNU 计划编写的 Unix Shell，是许多 Linux 发行版的默认 Shell，Bash 的命令语法是 Bourne Shell 命令语法的超集。数量庞大的 Bourne Shell 脚本大多不经修改即可在 Bash 中正常执行。

❑ 批处理: Batch，也称为批处理脚本。顾名思义，批处理就是对某些对象进行批量处理，通常被认为是一种简化的脚本语言，它一般应用于 DOS 和 Windows 系统中。批处理文件的扩展名为 bat。

❑ WMIC: WMIC 扩展 WMI (Windows Management Instrumentation, Windows 管理工具)，提供了对从命令行接口和批命令脚本执行系统管理的支持。

❑ 注册表: Registry，是 Microsoft Windows 中的一个重要的数据库，用于存储系统和应用程序的设置信息。

### 3.1.4 搬迁信息收集

俗话说“九层之台，起于垒土”，因此对于搬迁来讲，基础信息至关重要。我们需要准确掌握现有搬迁环境所涉及的所有物理与逻辑信息，才能更好地完成搬迁任务。因此搬迁的首要任务就是收集所有的资源信息。

#### 1. 计算资源信息收集

作为目标端新设备的配置依据，应收集（包含但不限于）如下信息：型号、序列号、CPU、MEM、硬件管理 IP、HBA WWN 号<sup>①</sup>。

编辑文件 /tmp/getESXInfo.sh，保存内容如下：

```
# 获取物理设备型号
xh=`esxcli hardware platform get | grep "Product Name" | awk -F: '{print $2}'`
# 获取物理设备序列号
SN=`esxcli hardware platform get | grep "Serial Number" | awk -F: '{print $2}'`
```

① 以上信息均通过 ESXCLI 指令进行收集，当然也可以通过 vCLI、PowerCLI 等不同的管理方式获取相关信息。

```

# 获取物理 CPU 个数
cpunum=`esxcli hardware cpu list | grep "CPU:" | wc -l`
# 获取物理内存容量 (以 GB 计算)
mem=`esxcli hardware memory get | grep "Physical Memory" | awk -F" " '{print $3}'`
TotalMem=`expr $mem / 1024 / 1024 / 1024`
# 获取硬件管理 IP
vmkip=`esxcli network ip interface ipv4 get | grep vmk0 | awk '{print $2}'`
# 获取 HBA WWN 信息
wwn=`esxcli storage core adapter list | grep link-up | awk -F: '{print $2}'`
| awk '{print $1}'`
wwn1=`echo $wwn | awk -F" " '{print $1}'`
wwn2=`echo $wwn | awk -F" " '{print $2}'`
echo $xh,$SN,$cpunum,$TotalMem,$vmkip,$wwn1,$wwn2 >/tmp/$vmkip.csv

```

可将以上脚本文件上传至所有 ESXI 系统, 以备批量收集信息之用。图 3-1 为显示结果。

```

/tmp # cat 10.10.10.29.csv
ProLiant BL685C G7, CNG, 64,255,10.10.10.29,1000a0b3ccea757,1000a0b3ccea75b

```

图 3-1 显示结果

## 2. 存储资源信息收集

基于现有 x86 环境对 ESXI 主机通过 FC 和 FCoE 协议分配外挂存储, 因此目标数据中心仍保持原有架构不变, 数据传输采用基于同构存储的 LUN COPY 方式。

首先需要确定哪些 LUN COPY 盘在目标端会挂给哪些 ESXI 主机, 需要收集的信息如下:

- ☐ 目标 ESXI IP
- ☐ 逻辑卷名
- ☐ 存储 naa 号 (WWN 号)

因为同一组中的外挂存储盘是通过共享的方式同时映射给同一群集内的多台 ESXI 主机的, 所以只需要对同一群集中的一台 ESXI 进行操作即可。

命令收集信息如下:

```

esxcli storage vmfs extent list | sort $1 | awk '{if($5==1){print $1 " ", $4}}'(注:
1 表明是外挂盘, 3 表明是内置盘)。

```

以下为同一群集内某台 ESXI 的外挂存储盘的逻辑卷名及存储号, 如图 3-2 所示。

```

3PAR_CNG2 2Z_31_30_2Y_LUN06,naa.50002ac016fe1df5
3PAR_CNG2 2Z_31_30_2Y_LUN07,naa.50002ac016ff1df5
3PAR_CNG2 2Z_31_30_2Y_LUN08,naa.50002ac017001df5
3PAR_CNG2 2Z_31_30_2Y_LUN09,naa.50002ac017011df5
3PAR_CNG2 2Z_31_30_2Y_LUN10,naa.50002ac017021df5
3PAR_CNG2 2Z_31_30_2Y_LUN11,naa.50002ac017031df5
3PAR_CNG2 2Z_31_30_2Y_LUN12,naa.50002ac017041df5
3PAR_CNG2 2Z_31_30_2Y_LUN13,naa.50002ac017051df5
3PAR_CNG2 2Z_31_30_2Y_LUN14,naa.50002ac017061df5
3PAR_CNG2 2Z_31_30_2Y_LUN15,naa.50002ac017071df5
3PAR_CNG2 2Z_31_30_2Y_LUN16,naa.50002ac017081df5

```

图 3-2 ESXI 的外挂存储盘逻辑卷名及存储号

### 3. 虚拟网络信息收集

虚拟网络信息的收集，对于在新环境中每台 ESXI 需要分配多少个逻辑上联口、多少 vSwitch 和网段 VLAN 的定义至关重要。

因此我们编辑以下 /tmp/getnetworkinfo.awk 文件，用于收集现有 ESXI 的 vSwitch Name、Uplinks（上联口）和 Portgroups 信息。

文件内容如下：

```
$1 ~ /Name:/ {print $1,$2}
$1 ~ /Uplinks:/ {print $0}
$1 ~ /Portgroups:/ {print $0}
```

我们编辑并执行以下 /tmp/getnetworkinfo.sh 文件，该文件会将 getnetworkinfo.awk 文件作为执行的过滤参数。

文件内容如下：

```
esxcli network vswitch standard list | awk -f /tmp/getnetworkinfo.awk
```

接下来就可以根据整理的各类信息完成后续逻辑迁移所需要的自动配置了。

## 3.2 利用 VMware 脚本简化虚拟化层的搬迁

我们需要在完成 ESXI 安装后执行如下虚拟化层的操作：

- (1) 为安装完成的 ESXI 开通 ESXI Shell 和远程 SSH 功能，确保可以批量管理 ESXI。
- (2) 打开 VMware 底层的 VMotion 功能以保证虚拟机的灵活迁移。
- (3) 为 ESXI 建立 vSwitch，并新建 port group（也就是网络交换机的概念，这种技术也称之为 VLAN），以保证注册的虚拟机节点能通过新建的虚拟网络与外部节点正常通信。
- (4) 批量挂载外挂存储，以确保虚拟机节点有可用的空间。
- (5) 注册虚拟机至 vCenter。
- (6) 将源端 vCenter 的目录结构复制到目标端 vCenter，并将虚拟机移动至指定的位置。
- (7) 在 vCenter 显示界面为每个虚拟机改名，并配置对应的 port group。
- (8) 启动虚拟机。

### 3.2.1 通过脚本完成 ESXI 安装后的基础设置

(1) 手工完成 ESXI 的安装后，请确认已经开启了 ESXI Shell 和 SSH，如图 3-3 所示。

我们可利用以下命令来开启 VMotion：

```
vim-cmd hostsvc/vmotion/vnic_set vmk0
```

VMotion 功能可以实现虚拟机人工漂移或自动漂移。开启这个功能之后，注册在同一台 ESXI 的虚拟机在开启时就可以漂移至同一群集的所有物理机上，从而实现均衡运行。



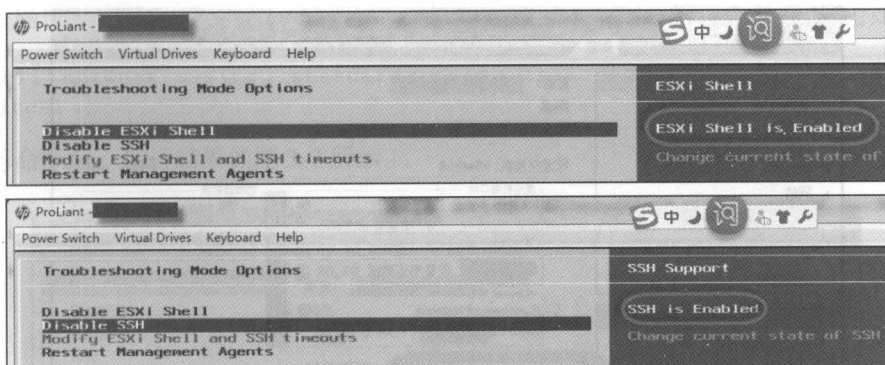


图 3-3 确认开启 ESXI Shell 和 SSH

(2) 网络配置 (利用收集的虚拟化层的信息, 根据规划完成 ESXI 的虚拟网络设置)。

由于公司网络规划采用了不同的区域, 不同区域设备的上联端口并不统一, 因此不适合使用 DVS vSwitch, 另外如果采用 Host Profile 绑定的方式, 一旦宿主机配置有改动, 就可能产生不合规提示。因此我决定根据规划, 在不同的区域通过批量脚本的方式建立 vSwitch 和绑定 port group

ESXI 安装后默认生成 vSwitch0 作为管理用的交换机, 我们需根据规划增加业务用的交换机, 举例如下。

(1) 添加名为 vSwitch1 的虚拟交换机, 可使用如下命令:

```
esxcli network vswitch standard add -v vSwitch1
```

或者使用如下命令:

```
esxcfg-vswitch -a vSwitch1
```



**注意** VMware 公司发布 vSphere 4 时引入了 ESXCLI 命令集, 在 vSphere5 时大大强化了 ESXCLI 指令的功能, 使其逐步替代类似于 esxcfg 的旧指令集。

执行命令后, 运行结果如图 3-4 所示。

(2) 查看上联口情况, 可使用如下命令:

```
esxcli network nic list
```

或者使用如下命令:

```
esxcfg-nics -l
```

执行命令后, 运行结果如图 3-5 所示。

(3) 将对应的物理网卡绑定到 vSwitch1 (vmnic2/vmnic3, 根据实际连线情况绑定到指定的物理网卡), 可使用如下命令:

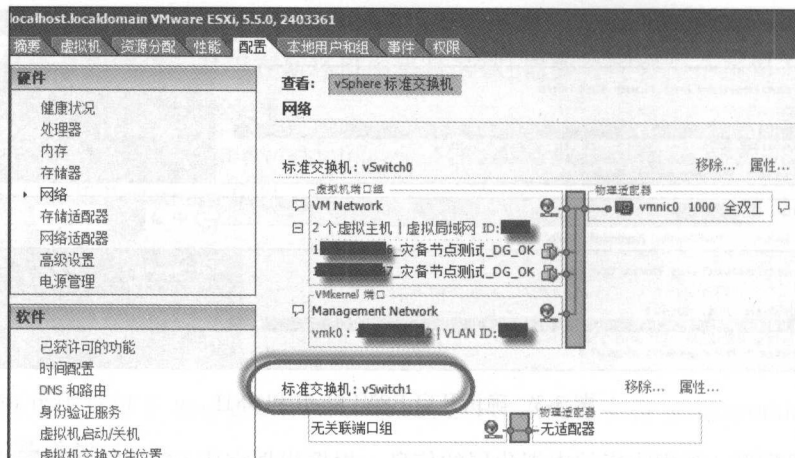


图 3-4 确认 ESXI 网络配置增加了 vSwitch1

```
~ # esxcli network nic list
```

Name	PCI Device	Driver	Link	Speed	Duplex	MAC Address	MTU	Description
vmnic0	0000:003:00.0	bnx2	up	1000	Full	a0:b3:cc:e8:48:b6	1500	Broadcom Corp
vmnic1	0000:003:00.1	bnx2	up	1000	Full	a0:b3:cc:e8:48:b8	1500	Broadcom Corp
vmnic2	0000:004:00.0	bnx2	down	0	Half	a0:b3:cc:e8:48:ba	1500	Broadcom Corp
vmnic3	0000:004:00.1	bnx2	down	0	Half	a0:b3:cc:e8:48:bc	1500	Broadcom Corp

图 3-5 确认 vmnic0 和 vmnic1 为活动的上联端口

```
esxcli network vswitch standard uplink add -u vmnic2(vmnic3) -v vSwitch1
```

或者使用如下命令：

```
esxcfg-vswitch -L vmnic2(vmnic3) vSwitch1
```

执行命令后，运行结果如图 3-6 所示。

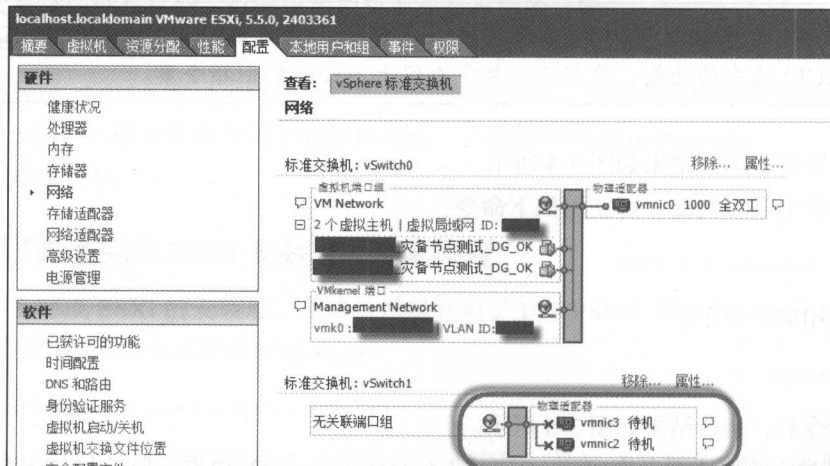


图 3-6 确认 vmnic2 和 vmnic3 被绑定至 vSwitch1，作为 vSwitch1 的上联端口

(4) 激活已添加在 vSwitch1 内的物理网卡 vmnic2/vmnic3 (双活模式), 可使用如下命令:

```
esxcli network vswitch standard policy failover set -a vmnic2,vmnic3 -v vSwitch1
```

执行命令后, 运行结果如图 3-7 所示。

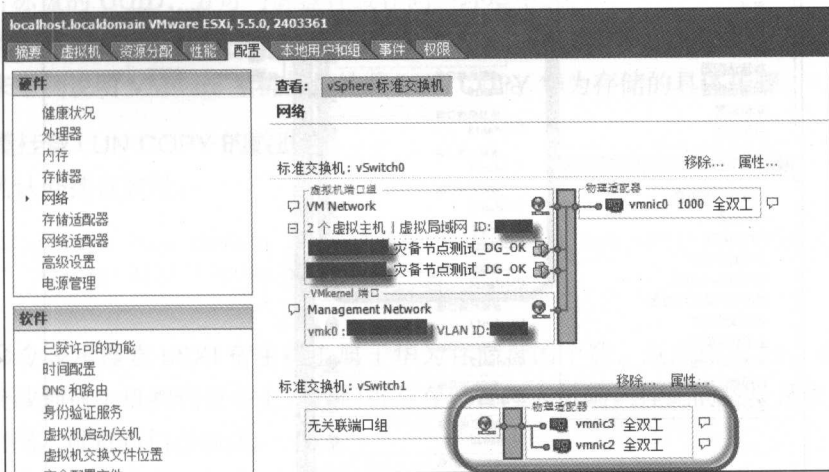


图 3-7 确认 vmnic2 和 vmnic3 作为双活策略被激活

(5) 添加名为 vlan1 到 vlan20 的 port group 到 vSwitch1, 可使用如下命令:

```
for i in `seq 1 20`;do esxcli network vswitch standard portgroup add -p vlan$i -v vSwitch1;done
```

或者使用如下命令:

```
for i in `seq 1 20`;do esxcfg-vswitch -A vlan$i vSwitch1;done
```

执行命令后, 运行结果如图 3-8 所示。

(6) 将对应 vlanid 为 1 到 20 的 vlan 与对应的 port group 绑定, 可使用如下命令:

```
for i in `seq 1 20`;do esxcli network vswitch standard portgroup set -p vlan$i -v $i;done
```

在同一 ESXi 主机下不同 vSwitch 中的 port group 是不能使用同一名字命名的, 因此以上 ESXCLI 的命令可直接针对指定的 port group 分配 vlanid, 而不必指明确切的 vSwitch。

或者使用如下命令:

```
for i in `seq 1 20`;do esxcfg-vswitch -v $i -p vlan$i vSwitch1;done
```

为减少循环, 提高效率可将以上语句合并如下:

```
for i in `seq 1 20`;do esxcfg-vswitch -A vlan$i vSwitch1;esxcfg-vswitch -v $i -p vlan$i vSwitch1;done
```

执行命令后，运行结果如图 3-9 所示。



图 3-8 确认建立了以 vlan1 到 vlan20 命名的 port group

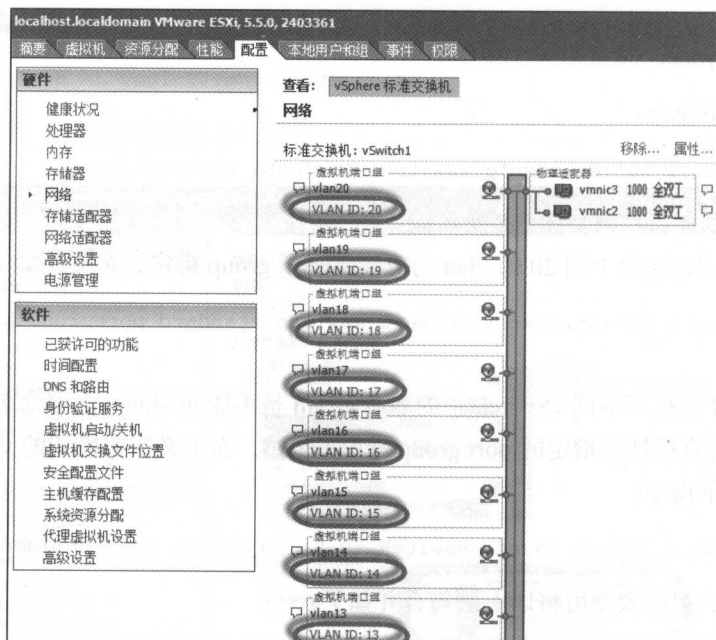


图 3-9 确认绑定了 vlan1 到 vlan20 的 ID 到对应的 port group

### 3.2.2 批量挂载数据盘

ESXI 对于生成的每个 VMFS DATASTORE 都存在一个唯一的 UUID, 当对其进行 LUN COPY 操作时, 原有的 UUID 也一并被复制到目标盘上。如果试图将目标盘挂载在与源盘同在一个的 vCenter 环境中时, 系统会显示 UUID 已存在, 从而导致挂载失败。只有通过重新签名更改目标盘的 UUID, 方可与源盘挂载在同一环境中。

因此我们事先在目标数据中心搭建了新的 vCenter 服务器。

下面来举例说明 VMware 主机批量挂载 LUN COPY 华为存储的具体步骤。

#### 1. 批量挂载 LUN COPY 的数据盘

##### (1) 确认外挂盘数量:

```
esxcli storage core device list | grep "HUAWEI" | awk -F "(" '{print $2}' | awk
-F ")" '{print $1}' | grep naa. | wc -l
50
```

以上命令表示搜索 ESXI 宿主机上属于华为存储盘的个数, 返回值为 50, 表明有 50 块外挂盘被挂载到宿主机物理设备上。我们可与存储管理人员确认挂载的数量是否一致, 如有不一致的情况, 可重新扫盘确认。

##### (2) 收集挂载设备的标识符:

```
esxcli storage vmfs snapshot list | grep "Volume Name" | awk -F ":" '{print
$2}' >/tmp/devices
```

以上命令将确认的 LUN COPY 属性盘的 WWN 号输出至 /tmp/devices 文件。当完成 LUN COPY 盘 WWN 号的收集后, 可将 devices 文件与存储管理人员提供的存储盘 WWN 号进行比对, 以此确认提取信息是否正确。

##### (3) 批量挂载 LUN COPY 盘:

```
for i in `cat /tmp/devices`;do esxcli storage vmfs snapshot mount -l $i;done
```

以上命令循环读取输出至 /tmp/devices 文件的外挂 LUN COPY 盘的 WWN 号, 并挂载这些 LUN COPY 盘至 ESXI 宿主机。

最终结果如图 3-10 所示。

标识	状态	设备	驱动器类型	容量
HUAWEI_214..._LUN44	正常	HUAWEI Fibre Ch...	非 SSD	1,023.75 G
HUAWEI_214..._LUN45	正常	HUAWEI Fibre Ch...	非 SSD	1,023.75 G
HUAWEI_214..._LUN46	正常	HUAWEI Fibre Ch...	非 SSD	1,023.75 G
HUAWEI_214..._LUN47	正常	HUAWEI Fibre Ch...	非 SSD	1,023.75 G
HUAWEI_214..._LUN48	正常	HUAWEI Fibre Ch...	非 SSD	1,023.75 G
HUAWEI_214..._LUN49	正常	HUAWEI Fibre Ch...	非 SSD	1,023.75 G
HUAWEI_214..._LUN50	正常	HUAWEI Fibre Ch...	非 SSD	1,023.75 G

图 3-10 显示批量挂载的 LUN COPY 盘



**注意** 存储挂载名称建议根据“存储品牌名\_主机序列号\_LUN 号”命名。

以上操作只完成了对同一群集中某台主机的挂盘操作，我们只需要在同一群集的其他主机上通过执行扫描光纤链路的操作，即可让这些外挂盘被群集中的其他主机识别。具体步骤如下：配置→存储器→全部重新扫描。或者使用如下命令：

```
esxcli storage core adapter rescan
```

执行命令，运行结果如图 3-11 所示。

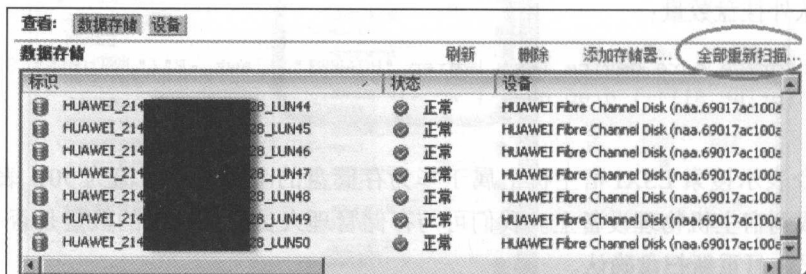


图 3-11 显示同群集中其他主机批量挂载的 LUN COPY 盘

## 2. 批量挂载新分配的盘

下面将举例说明为 VMware 主机批量挂载 LUN COPY 华为存储新盘的具体步骤<sup>①</sup>。

### (1) 确认新分配外挂盘的数量：

```
esxcli storage core device list | grep "HUAWEI" | awk -F "(" '{print $2}' | awk -F ")" '{print $1}' | grep naa. | wc -l
50
```

以上命令表示搜索 ESXI 宿主机上属于华为存储盘的个数，返回值为 50，表明有 50 块外挂盘被挂载到宿主机物理设备上。我们可与存储管理人员确认挂载的数量是否一致，如有不一致的情况，可重新扫盘确认。

### (2) 收集新分配外挂盘的标识符：

```
esxcli storage core device list | grep "HUAWEI" | awk -F "(" '{print $2}' | awk -F ")" '{print $1}' | grep naa. >>/tmp/newdevice
naa.69017ac100aec92a7f8bfc7800000021
naa.69017ac100aec92a7f8bffa10000002a
naa.69017ac100aec92a7f8bfce200000022
.....(省略中间的若干块盘)
naa.69017ac100aec92a7f8bf85100000013
naa.69017ac100aec92a7f8bf54e0000000a
naa.69017ac100aec92a7f8bf6b30000000e
```

① 此步骤只针对新采购资源，如需对已使用的资源添加新分配外挂盘，则需要在 ESXI 上利用 diff 进行比对。



以上命令只将新分配盘的 WWN 号输出至 /tmp/newdevice 文件。当完成新分配外挂盘 WWN 号的收集后, 可将 newdevices 文件与存储管理人员提供的存储盘 WWN 号进行比对, 以此确认提取信息是否正确。

(3) 通过列编辑软件, 将新挂盘名称列加入 /tmp/newdevice 文件。接着就可通过列编辑软件在 WWN 号所属的列前增加一行, 用于标记新分配盘的名称, 格式如下:

```
HUAWEI_ 214????26_ 214????28_LUN01,naa.69017ac100aec92a7f8bfc7800000021
HUAWEI_ 214????26_ 214????28_LUN02,naa.69017ac100aec92a7f8bffa10000002a
HUAWEI_ 214????26_ 214????28_LUN03,naa.69017ac100aec92a7f8bfce200000022
.....(省略中间的若干块盘)
HUAWEI_ 214????26_ 214????28_LUN48,naa.69017ac100aec92a7f8bf85100000013
HUAWEI_ 214????26_ 214????28_LUN49,naa.69017ac100aec92a7f8bf54e0000000a
HUAWEI_ 214????26_ 214????28_LUN50,naa.69017ac100aec92a7f8bf6b30000000e
```

也可以远程连接至 ESXI 的 CONSOLE 界面, 执行以下命令更新 /tmp/newdevice 文件的内容:

```
cat > /tmp/newdevice <<EOF
HUAWEI_ 214????26_ 214????28_LUN01,naa.69017ac100aec92a7f8bfc7800000021
HUAWEI_ 214????26_ 214????28_LUN02,naa.69017ac100aec92a7f8bffa10000002a
HUAWEI_ 214????26_ 214????28_LUN03,naa.69017ac100aec92a7f8bfce200000022
.....(省略中间的若干块盘)
HUAWEI_ 214????26_ 214????28_LUN48,naa.69017ac100aec92a7f8bf85100000013
HUAWEI_ 214????26_ 214????28_LUN49,naa.69017ac100aec92a7f8bf54e0000000a
HUAWEI_ 214????26_ 214????28_LUN50,naa.69017ac100aec92a7f8bf6b30000000e
EOF
```

通过以上操作, 我们定义了新分配盘的盘名和 WWN 号, 作为后续新加盘命令的参数。

(4) 批量为新增盘分区 (以 1TB 和 2TB 的格式为例):

```
for i in `cat /tmp/newdevice | awk -F"," '{print $2}'`; do partedUtil setptbl
/vmfs/devices/disks/$i gpt "1 2048 2147483614
AA31E02A400F11DB9590000C2911D1B8 0"; done      (按 1TB 盘的格式)

for i in `cat /tmp/newdevice | awk -F"," '{print $2}'`; do partedUtil setptbl
/vmfs/devices/disks/$i gpt "1 2048 4294961684
AA31E02A400F11DB9590000C2911D1B8 0"; done      (按 2TB 盘的格式)
```

以上命令分别按 1TB 和 2TB 的参数, 对新盘进行了分区。

(5) 批量新建 VMFS5 文件系统并自动挂盘:

```
for i in `cat /tmp/newdevice`;do name=`echo $i | awk -F"," '{print $1}'`
vid=`echo $i | awk -F"," '{print $2}'`; vmkfstools -C vmfs5 -S $name -b 1m
/vmfs/devices/disks/$vid:1 ;done
```

以上命令对新盘进行了格式化操作, 将其格式化为 VMFS5 的文件系统后, 系统自动进行了挂盘操作。

如图 3-12 所示为正在挂盘操作的示例图。



图 3-12 正在执行新盘的挂载

最终结果如图 3-13 所示：

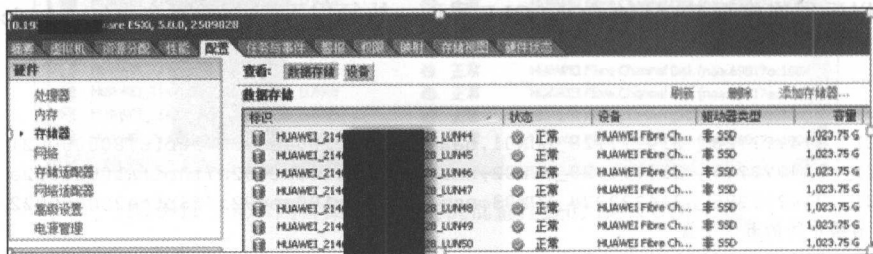


图 3-13 显示批量挂载的新分配盘



**注意** 存储挂载的名称请以“存储品牌名\_主机序列号\_LUN号”命名。

以上操作只完成了对同一群集中某台主机的挂盘操作，我们只需要在同一群集的其他主机上通过执行扫描光纤链路的操作，即可让这些外挂盘被群集中的其他主机识别。具体步骤如下：配置→存储器→全部重新扫描。或者使用如下命令：

```
esxcli storage core adapter rescan
```

执行命令，运行结果如图 3-14 所示。

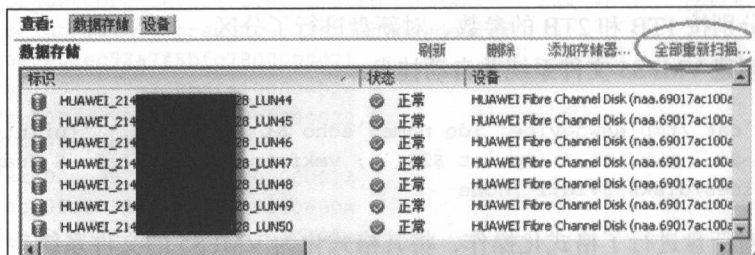


图 3-14 显示同群集中其他主机批量挂载的新分配盘

### 3.2.3 批量注册虚拟机

3.2.2 节中，我们已将 50 块 LUN COPY 的外挂盘挂给了一组刀片式服务器，接下来我们需要将所涉及的该批次的所有虚拟机注册进 vCenter Server。

对群集中的第一台 ESXI 执行以下脚本，将所有虚拟机的配置文件 .vmx 导出到一个文件中。

```
find /vmfs/volumes/ -name *.vmx >/tmp/vmlist
```

为避免开机时因 VM UUID 重复而产生如图 3-15 所示的交互式问答的提示，我们可以执行如下命令，批量删除 VMX 中的 uuid.bios 和 uuid.location，使虚拟机重新产生新的 UUID：

```
for i in `cat /tmp/vmlist` ; do sed -i '/^uuid.bios/d' $i ; done
for i in `cat /tmp/vmlist` ; do sed -i '/^uuid.location/d' $i ; done
```

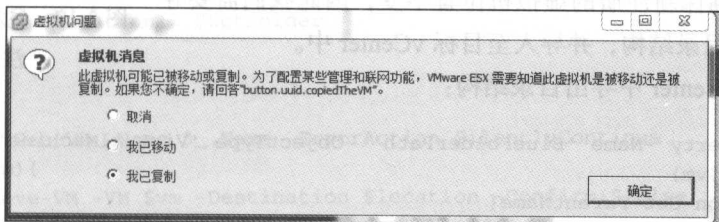


图 3-15 因 VM UUID 重复而产生的提示

为避免虚拟机注册后遗留下如图 3-16 所示的“影子”网段信息，我们可执行以下命令，批量删除 VMX 中的 ethernet0.networkName，使虚拟机去除老旧的网络信息：

```
for i in `cat /tmp/vmlist` ; do sed -i '/^ethernet0.networkName/d' $i ; done
```

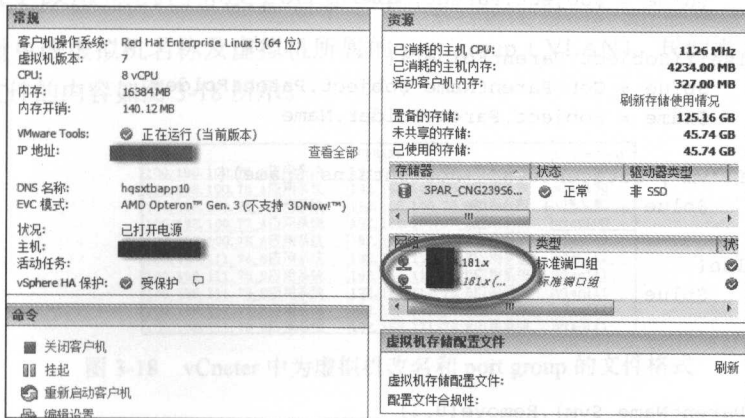


图 3-16 “影子”网段情况

以上是对 VMX 配置文件的信息修改，必须在将虚拟机注册进 vCenter 之前完成。如果

先将虚拟机注册进 vCenter，然后再执行以上操作，则不会有任何效果。

运行如下命令，注册虚拟机：

```
IFS=$(echo -en "\n\b")
for i in `cat /tmp/vmlist` ; do vim-cmd solo/registervm $i ; done
```

在注册的同时，我们会看见命令行窗口的数字正在逐渐增加，该数字表明为注册入 vCenter Server 的虚拟机分配的 VMID，图 3-17 显示有 1004 个虚拟机完成了注册：

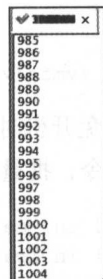


图 3-17 显示 1004 个虚拟机注册成功

### 3.2.4 vCenter 目录结构的调整

我们需要将源 vCenter 的目录结构原封不动地挪移到目标 vCenter 中，以确保新注册的虚拟机位置不变，因此我们需要导出源 vCenter 的目录结构，并导入至目标 vCenter 中。

(1) 从源 vCenter 中导出目录结构：

```
New-VIProperty -Name 'BlueFolderPath' -ObjectType 'VirtualMachine' -Value {
    param($vm)
    function Get-ParentName{
        param($object)
        if($object.Folder){
            $blue = Get-ParentName $object.Folder      # 获取路径
            $name = $object.Folder.Name                 # 获取 vm_name
        }
        elseif($object.Parent -and $object.Parent.GetType().Name -like
            "Folder*"){
            $blue = Get-ParentName $object.Parent
            $name = $object.Parent.Name
        }
        elseif($object.ParentFolder){
            $blue = Get-ParentName $object.ParentFolder
            $name = $object.ParentFolder.Name
        }
        if("vm", "Datacenters" -notcontains $name){
            $blue + "/" + $name
        }
        else{
            $blue
        }
    }

    (Get-ParentName $vm).Remove(0,1)
} -Force | Out-Null
$dcName = "zbvc00-vcenter"
```

```
Get-VM -Location (Get-Datacenter -Name $dcName) | Select Name, BlueFolderPath
```

```
| Export-Csv "C:\vm-folder.csv" -NoTypeInfoInformation -UseCulture -encoding utf8
```

(2) 将目录结构导入目标 vCenter 中 (并将虚拟机移动至指定目录路径下):

```
$newDatacenter = "zbvc00-vcenter"
$startFolder = get-Folder -Name vm -Location (Get-Folder -Name vm -Location
(Get-Datacenter -Name $newDatacenter))

Import-Csv "C:\vm-folder.csv" -UseCulture | %{
    $location = $startFolder
    $_.BlueFolderPath.TrimStart('/').Split('/') | %{
        $tgtFolder = Get-Folder -Name vm -Location $location -ErrorAction
        SilentlyContinue
        if(!$tgtFolder){
            $location = New-Folder -Name $_ -Location $location
        }
        else{
            $location = $tgtFolder
        }
    }

    $vm = Get-VM -Name $_.Name -ErrorAction SilentlyContinue
    if($vm){
        Move-VM -VM $vm -Destination $location -Confirm:$false
    }
}
```



注意 以上脚本截取自 VMware 社区论坛, 并根据实际情况修改而成。

### 3.2.5 批量更改虚拟机名称及 port group

为了批量更改虚拟机名称及虚拟机所属的 port group (VLAN), 我们需要建立一个参数文件, 参数文件的内容如图 3-18 所示。

oldname	NEWNAME	portgroup
100.198.100.74_A应用系统	192.168.100.74_A应用系统	vlan100
100.198.100.75_A应用系统	192.168.100.75_A应用系统	vlan100
100.198.100.76_A应用系统	192.168.100.76_A应用系统	vlan100
100.198.100.77_A应用系统	192.168.100.77_A应用系统	vlan100
100.198.100.78_A应用系统	192.168.100.78_A应用系统	vlan100
100.198.111.74_B应用系统	192.168.111.74_B应用系统	vlan111
100.198.111.75_B应用系统	192.168.111.75_B应用系统	vlan111
100.198.111.76_B应用系统	192.168.111.76_B应用系统	vlan111
100.198.111.77_B应用系统	192.168.111.77_B应用系统	vlan111
100.198.111.78_B应用系统	192.168.111.78_B应用系统	vlan111

图 3-18 vCenter 中为虚拟机改名和 port group 的文件格式

我们将参数文件保存为 c:\zbportgroup.csv, 由于主机名采用的是中文字符, 因此保存参数文件时编码务必采用 UTF-8 字符集, 以确保 PowerCLI 中可以正常识别。

接下来登录 PowerCLI 控制机, 连接入指定的 vCenter Server, 如图 3-19 所示。

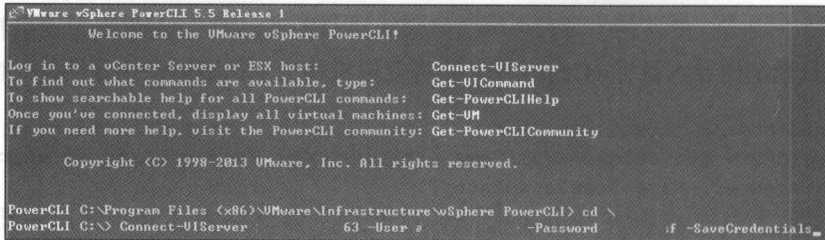


图 3-19 使用 Connect-VIServer 登录 PowerCLI 管理界面

导入参数文件：

```
$vmlist=Import-CSV "c:\zbpportgroup.csv"
```

为虚拟机重命名：

```
FOREACH ($vm in $vmlist)
{ Get-VM $($vm.oldname) | Set-vm -name $($vm.NEWNAME) -confirm:$false |
Out-File -Append changname.txt }
```

为虚拟机重新设置 port group：

```
FOREACH ($vm in $vmlist)
{ Get-vm $($vm.NEWNAME) | get-networkadapter | set-networkadapter -networkname
$($vm.portgroup) -confirm:$false | Out-File -Append changportgroup.txt }
```

为减少循环以提高效率，可将以上语句合并如下：

```
FOREACH ($vm in $vmlist)
{ (Get-VM $($vm.oldname) | Set-vm -name $($vm.NEWNAME) -confirm:$false) -and
(Get-vm $($vm.NEWNAME) | get-networkadapter | set-networkadapter -networkname
$($vm.portgroup) -confirm:$false) }
```

正在执行批量更新 port group 的虚拟机，如图 3-20 所示。

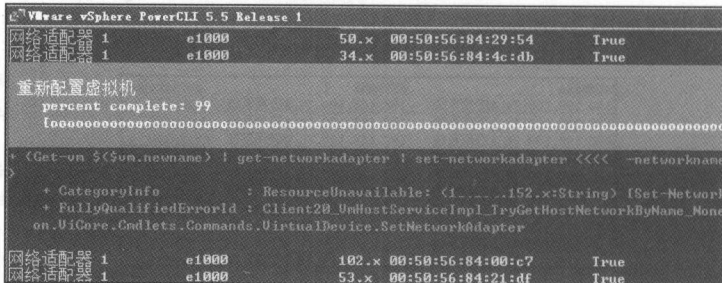


图 3-20 执行更改 VM name 和 port group 的操作

### 3.2.6 批量设置虚拟机版本和 CPU、内存保留值

因搬迁的环境中虚拟机版本不统一，既有 version 4 的，又有 version 7 和 8 的虚拟机，



为保证虚拟机的版本统一，故需将虚拟机版本调整为 version 8，同时为保证群集中所有的虚拟机都能顺利开启，还需要将 CPU 和内存的预留资源调整为 0，命令如下：

```
foreach ($vm in $vmlist)
{
    set-vm $vm.oldname -version v8 -confirm:$false;
    (Get-VM $vm.oldname | Get-VMResourceConfiguration | where {$_ .CPULimitMhz -ne '0'}) | Set-VMResourceConfiguration -cpuReservationMhz 0);
    (Get-VM $vm.oldname | Get-VMResourceConfiguration | where {$_ .MemReservationMB -ne '0'}) | Set-VMResourceConfiguration -MemReservationMB 0)
}
```

批量启动虚拟机

```
FOREACH ($vm in $vmlist)
{start-vm -VM $vm }
```

### 3.3 利用批处理与 Shell 脚本简化逻辑节点的搬迁

#### 3.3.1 逻辑节点切换脚本的思路

众所周知，对于服务器的搬迁，不只会涉及物理层面的设备搬迁。在物理设备搬迁到新机房后，往往还需要对虚拟机中的多种参数进行相关调整。

如果管理的服务器台数比较少的话，则可以采用手工设置的方式逐一更改参数。但是，如果手中管理的 Windows 和 Linux 虚拟机有数千台的话，那么这些繁琐的参数调整一定会让你头疼不已。其中需要调整的参数如下：

- ☐ IP
- ☐ GATEWAY
- ☐ DNS
- ☐ WSUS
- ☐ NTP
- ☐ HOSTS

也许大家要问，像 DNS、WSUS、NTP 这类服务器地址是有固定 IP 的，设置起来应该相对容易，但像 IP 地址和 GATEWAY 怎么办呢？下面就来讲述下如何编写这个脚本。

服务器条件如下：

- ☐ 搬迁环境涉及两类资源，一类为容灾环境节点、一类为研发测试环境节点。
- ☐ 每台服务器都利用的是内网网段 IP，通过 NAT 转换到公网。
- ☐ 每台服务器都只有一个本地连接的网口。
- ☐ 所有网段的网关地址的最后一位为 254（基于 C 类网段进行分配）。

由于新注册到目标数据中心的所有节点，在开启后仍然保持了源端 LUN COPY 节点的

所有参数。而且并非所有节点都安装了适合虚拟机版本的 VMware-Tools 组件，因此无法统一通过 PowerCLI 的 `invoke-vmscript` 命令直接对虚拟机发送指令进行操作。结合工作中的实际需求，我考虑在虚拟机未断开复制之前对源端主机预先上传切换用的脚本，搬迁至目标端之后，人工执行切换脚本并选取对应的环境，系统再根据人工反馈的环境值，自动更改对应的参数来完成虚拟机内所有参数的切换工作。

每台服务器都有一个搬迁前的在用 IP 地址，另外根据规划已知要在搬迁后为每台服务器分配一个新的 IP 地址。因此可根据项目规划的要求制作一张新旧 IP 切换对照表，俗称字典文件，该对照表包含如下内容：

第一列为原有的旧 IP，第二列为搬迁后的新 IP，第三列为搬迁后需要调整的新主机名（如无需调整主机名，则第三列可忽略。Linux 的字典文件用 TAB 作为分隔符，Windows 的字典文件用空格作为分隔符），该文件将最终随同切换用脚本文件在搬迁之前一并上传至逻辑节点的指定目录中待用。

以下表 3-2 仅为参考示例。

表 3-2 新旧 IP 切换对照表

现有 IP	新 IP	新主机名
100.198.100.71	192.168.100.71	ser1
100.198.100.72	192.168.100.72	ser2
100.198.100.73	192.168.100.73	ser3
100.198.100.74	192.168.100.74	ser4
100.198.100.75	192.168.100.75	ser5
100.198.111.76	192.168.111.76	ser6
100.198.111.77	192.168.111.77	ser7
100.198.111.78	192.168.111.78	ser8
100.198.111.79	192.168.111.79	ser9
100.198.111.80	192.168.111.80	ser10

获取目前主机上的 IP 地址情况，然后与以上字典文件中的现有 IP 列的 IP 地址进行比对，如获取到对应的现有 IP，则再提取与现有 IP 同一行中对应的新 IP 和新主机名。并将新 IP 的值自动赋予 Windows 的网卡，将新的主机名赋予操作系统（如果有需要更改主机名的情况），这样就完成了 IP 的自动替换和主机名更改的工作。

3.3.2 利用批处理脚本简化 Windows 逻辑节点的搬迁

下面是基于 Windows 批处理的脚本，主要实现功能为判断操作系统（Win2003/Win2008）和应用环境（研发和容灾），执行不同的批处理指令，将字典文件中的旧 IP 切换为对应的新 IP：

```
@echo ++++++
```

```

@echo + 欢迎使用 Win2003/Win2008 切换脚本, 请根据切换环境输入指定的参数! +
@echo ++++++
@choice /C:123 /N /M "1: 研发测试环境 2: 容灾环境 3: 退出"

if errorlevel 3 goto end
if errorlevel 2 goto zb
if errorlevel 1 goto kfcs

# 以上语句显示欢迎页面, 同时还显示应用环境菜单, 根据操作的选取, 可跳转到不同的环境
:kfcs                                     # (设置开发测试环境的所有参数)
set adapter=                             # (初始化网卡变量)
set oldip=                               # (初始化旧 IP 变量)
set newip=                               # (初始化新 IP 变量)
set gateway=                             # (初始化新网关变量)
set dns1=                                # (初始化主 DNS 变量)
set dns2=                                # (初始化辅 DNS 变量)
set mask=255.255.255.0                   # (设置子网掩码变量为 255.255.255.0)
set wsusserver=                           # (初始化 WSUS 服务器变量)
set ntpserver=                           # (初始化 NTP 服务器变量)
set wsusserver=http://192.168.127.198     # (为 WSUS 服务器赋予指定 IP)
set ntpserver="192.168.127.103"           # (为 NTP 服务器赋予指定 IP)
goto ver

:zb                                     # (设置灾备环境的所有参数)
set adapter=                             # (初始化网卡变量)
set oldip=                               # (初始化旧 IP 变量)
set newip=                               # (初始化新 IP 变量)
set gateway=                             # (初始化新网关变量)
set dns1=192.168.127.1                   # (为 DNS1 服务器赋予指定 IP)
set dns2=192.168.127.200                 # (为 DNS2 服务器赋予指定 IP)
set mask=255.255.255.0                   # (设置子网掩码变量为 255.255.255.0)
set wsusserver=                           # (初始化 WSUS 服务器变量)
set ntpserver=                           # (初始化 NTP 服务器变量)
set wsusserver=http://192.168.16.54       # (为 WSUS 服务器赋予指定 IP)
set ntpserver="192.168.16.54"             # (为 NTP 服务器赋予指定 IP)
goto ver

:ver                                     # (判断 Windows 版本是 2003 还是 2008)
ver | find /i "6.1." > NUL
if %errorlevel% equ 0 (goto win2008)
ver | find /i "5.2." > NUL
if %errorlevel% equ 0 (goto win2003)

:win2008                                # (如果是 Win2008, 则执行如下语句)
ipconfig | findstr /i "以太网适配器">c:\tmp\ipchange\adapter.txt
for /f "tokens=2*" %%i in (c:\tmp\ipchange\adapter.txt) do
@echo %%i %%j>c:\tmp\ipchange\adapter.txt
for /f "tokens=1 delims=:" %%i in (c:\tmp\ipchange\adapter.txt) do (
set adapter=%%i
echo %%i>c:\tmp\ipchange\adapter.txt
)

```

# 以上语句保存以太网适配器名称

```
ipconfig | findstr /i "IPv4">c:\tmp\ipchange\oldip.txt
for /f "tokens=2 delims=:" %i in (c:\tmp\ipchange\oldip.txt) do
@echo %i>c:\tmp\ipchange\oldip.txt
for /f "tokens=" %i in (c:\tmp\ipchange\oldip.txt) do
@echo %i>c:\tmp\ipchange\oldip.txt
for /f %i in (c:\tmp\ipchange\oldip.txt) do set oldip=%i
# 以上语句保存旧 IP
```

rem 获取 DNS 并保存

```
ipconfig /all | findstr /C:"DNS Servers" /C:"DNS 服务器"
">c:\tmp\ipchange\olddns.txt
for /f "tokens=2 delims=:" %i in (c:\tmp\ipchange\olddns.txt) do @echo %i
| findstr "^[0-9]*.[0-9]*.[0-9]*.[0-9]">c:\tmp\ipchange\olddns.txt
for /f "tokens=" %i in (c:\tmp\ipchange\olddns.txt) do
@echo %i>c:\tmp\ipchange\olddns.txt
# 以上语句保存旧 DNS 服务器, 以备更换后有旧 DNS 的回溯
```

```
findstr /i "%oldip%"
```

```
c:\tmp\ipchange\ipcheck.txt>c:\tmp\ipchange\oldtonewip.txt
for /f "tokens=2" %i in (c:\tmp\ipchange\oldtonewip.txt) do
@echo %i>c:\tmp\ipchange\newip.txt
for /f %i in (c:\tmp\ipchange\newip.txt) do set newip=%i
# 以上语句查找字典文件 ipcheck.txt 中是否存在旧 IP 的条目, 如果存在则提取与旧 IP 对应
的新 IP 值, 并赋给 newip 变量
```

```
for /f "tokens=1-3 delims=." %i in (c:\tmp\ipchange\newip.txt) do
echo %i.%j.%k.254>c:\tmp\ipchange\gatewayip.txt
for /f %i in (c:\tmp\ipchange\gatewayip.txt) do set gatewayip=%i
# 以上语句提取 newip 变量中 以 . 为分隔符的前三位, 最后一位设置为 254, 作为
GATEWAY 的新值, 并赋给 gatewayip 变量
```

rem 备份 hosts 文件, 并根据字典文件的定义, 替换 hosts 内关联节点对应的 IP

```
setlocal enabledelayedexpansion
for /f "tokens=" %i in (C:\WINDOWS\system32\drivers\etc\hosts) do (set
var=%i
set "var=!var:%oldip%=%newip%!"
echo !var! >> C:\WINDOWS\system32\drivers\etc\hosts.new
)
ren C:\WINDOWS\system32\drivers\etc\hosts hosts.old
copy C:\WINDOWS\system32\drivers\etc\hosts.old C:\tmp\ipchange
ren C:\WINDOWS\system32\drivers\etc\hosts.new hosts
endlocal
# 以上语句备份 c:\windows\system32\drivers\etc\hosts 文件, 并根据字典文件的定义替换
hosts 文件内关联节点对应的 IP 与主机名的映射关系
```

rem 自动设置新 IP 和网关

# (rem 为脚本中的注释语句)

rem 自动设置新 DNS

# (rem 为脚本中的注释语句)

```
netsh interface ipv4 set address "%adapter%" static %newip% %mask%
```

```

gateway=%gatewayip% 1
netsh interface ipv4 del dnsservers name="%adapter%" all
netsh interface ipv4 set dnsservers name="%adapter%" source=static %dns1%
register=primary validate=no
# netsh interface ipv4 add dnsservers "%adapter%" %dns2% index=2 validate=no
# 以上语句将先前几个步骤获取到的 adapter、newip、gateway、dns1、dns2 的变量，作为设置
  的参数直接为 Windows 系统设置新 IP、新网关和新 DNS
goto envir # (跳转到 envir，调整其他附加环境)

:win2003 # (如果是 Win2003，则执行如下语句)
ipconfig | findstr /i "Ethernet adapter">c:\tmp\ipchange\adapter.txt
for /f "tokens=3* delims=" %i in (c:\tmp\ipchange\adapter.txt) do
@echo %i %j>c:\tmp\ipchange\adapter.txt
for /f "tokens=1 delims=" %i in (c:\tmp\ipchange\adapter.txt) do (
set adapter=%i
echo %i>c:\tmp\ipchange\adapter.txt
)
# 以上语句保存以太网适配器的名称
ipconfig | findstr /i "IP Address">c:\tmp\ipchange\oldip.txt
for /f "tokens=2 delims=" %i in (c:\tmp\ipchange\oldip.txt) do
@echo %i>c:\tmp\ipchange\oldip.txt
for /f "tokens=*" %i in (c:\tmp\ipchange\oldip.txt) do
@echo %i>c:\tmp\ipchange\oldip.txt
for /f %i in (c:\tmp\ipchange\oldip.txt) do set oldip=%i
# 以上语句保存旧 IP

rem 获取 DNS 并保存
ipconfig /all | findstr /C:"DNS Servers" /C:"DNS 服务器"
">c:\tmp\ipchange\olddns.txt
for /f "tokens=2 delims=" %i in (c:\tmp\ipchange\olddns.txt) do @echo %i
| findstr "[0-9]*.[0-9]*.[0-9]*.[0-9]">c:\tmp\ipchange\olddns.txt
for /f "tokens=*" %i in (c:\tmp\ipchange\olddns.txt) do
@echo %i>c:\tmp\ipchange\olddns.txt
# 以上语句保存旧 DNS 服务器，以备更换后有旧 DNS 的追溯

findstr /i "%oldip%>"
c:\tmp\ipchange\ipcheck.txt>c:\tmp\ipchange\oldtonewip.txt
for /f "tokens=2" %i in (c:\tmp\ipchange\oldtonewip.txt) do
@echo %i>c:\tmp\ipchange\newip.txt
for /f %i in (c:\tmp\ipchange\newip.txt) do set newip=%i
# 以上语句查找字典文件 ipcheck.txt 中是否存在旧 IP 的条目，如果存在则提取与旧 IP 对应
  的新 IP 值，并赋给 newip 变量

for /f "tokens=1-3 delims=" %i in (c:\tmp\ipchange\newip.txt) do
echo %i.%j.%k.254>c:\tmp\ipchange\gatewayip.txt
for /f %i in (c:\tmp\ipchange\gatewayip.txt) do set gatewayip=%i
# 以上语句提取 newip 变量中，以 . 为分隔符的前三位，最后一位设置为 254，作为
  GATEWAY 的新值，并赋给 gatewayip 变量

rem 备份并替换 hosts 文件

```

```

setlocal enabledelayedexpansion
for /f "tokens=*" %i in (C:\WINDOWS\system32\drivers\etc\hosts) do (set
var=%i
set "var=!var:%oldip%=%newip%!"
echo !var! >> C:\WINDOWS\system32\drivers\etc\hosts.new
)
ren C:\WINDOWS\system32\drivers\etc\hosts hosts.old
copy C:\WINDOWS\system32\drivers\etc\hosts.old C:\tmp\ipchange
ren C:\WINDOWS\system32\drivers\etc\hosts.new hosts
endlocal
# 以上语句备份 c:\windows\system32\drivers\etc\hosts 文件，并根据字典文件的定义替换
hosts 文件内关联节点对应的 IP 与主机名的映射关系

rem 自动设置新 IP 和网关
rem 自动设置新 DNS
netsh interface ip set address name="%adapter%" source=static %newip% %mask%
gateway=%gatewayip% auto
netsh interface ip del dns name="%adapter%" all
netsh interface ip set dns name="%adapter%" source=static %dns1%
register=primary
#netsh interface ip add dns "%adapter%" %dns2% index=2
# 以上语句将先前几个步骤获取到的 adapter、newip、gateway、dns1、dns2 的变量，作为设置
的参数直接为 Windows 系统设置新 IP、新网关和新 DNS
goto envir                                # (跳转到 envir，调整其他附加环境)

:envir                                    # (执行调整其他附加环境的语句)

rem =====
rem 2. 设置 Patrol 参数
echo Windows Registry Editor Version 5.00>c:\tmp\ipchange\patrolagent.reg
echo.>>c:\tmp\ipchange\patrolagent.reg
echo
[HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Services\PatrolAgent\Parameters]>>c:\tmp\ipchange\patrolagent.reg
echo "Port"=dword:00000c6d>>c:\tmp\ipchange\patrolagent.reg
echo "ID"="%newip%">>c:\tmp\ipchange\patrolagent.reg
regedit /s c:\tmp\ipchange\patrolagent.reg
# 以上语句将新 IP 写入 PatrolAgent 的注册表参数，确保 Patrol 监控软件正常工作

rem=====
rem 3. 设置 WSUS 服务器参数
echo Windows Registry Editor Version 5.00>c:\tmp\ipchange\wsus.reg
echo.>>c:\tmp\ipchange\wsus.reg
echo
[HKEY_LOCAL_MACHINE\SOFTWARE\Policies\Microsoft\windows\WindowsUpdate]>>c:
\tmp\ipchange\wsus.reg
echo "WUServer"=%wsusserver%>>c:\tmp\ipchange\wsus.reg
echo "WUStatusServer"=%wsusserver%>>c:\tmp\ipchange\wsus.reg
echo.>>c:\tmp\ipchange\wsus.reg
echo

```



```
[HKEY_LOCAL_MACHINE\SOFTWARE\Policies\Microsoft\windows\WindowsUpdate\AU]>
>c:\tmp\ipchange\wsus.reg
echo "NoAutoUpdate"=dword:00000000>>c:\tmp\ipchange\wsus.reg
echo "AUOptions"=dword:00000003>>c:\tmp\ipchange\wsus.reg
echo "ScheduledInstallDay"=dword:00000000>>c:\tmp\ipchange\wsus.reg
echo "ScheduledInstallTime"=dword:00000017>>c:\tmp\ipchange\wsus.reg
echo "UseWUServer"=dword:00000001>>c:\tmp\ipchange\wsus.reg
regedit /s c:\tmp\ipchange\wsus.reg
# 以上语句将 WSUS 的 IP 及配置参数写入注册表, 确保 Windows 客户端能从 WSUS 正确获取
更新补丁
```

```
rem=====
rem 4. 设置 NTP 服务器参数
echo Windows Registry Editor Version 5.00>c:\tmp\ipchange\ntp.reg
echo.>>c:\tmp\ipchange\ntp.reg
echo
[HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Windows\CurrentVersion\DateTime\Ser
vers]>>c:\tmp\ipchange\ntp.reg
echo @="0">>c:\tmp\ipchange\ntp.reg
echo "0"=%ntpserver%>>c:\tmp\ipchange\ntp.reg
echo
[HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\services\W32Time\TimeProvider
s\NtpClient]>>c:\tmp\ipchange\ntp.reg
echo "Enabled"=dword:00000001>>c:\tmp\ipchange\ntp.reg
echo "SpecialPollInterval"=dword:2a300>>c:\tmp\ipchange\ntp.reg
regedit /s c:\tmp\ipchange\ntp.reg
```

```
w32tm /config /manualpeerlist:%ntpserver% /syncfromflags:manual
/reliable:yes
w32tm /config /update
net stop w32time
net start w32time
w32tm /resync
sc config w32time start= auto
# 以上语句将 NTP 的 IP 及配置参数写入注册表, 确保 Windows 客户端定期与 NTP 服务器同步
```

```
rem=====
rem 5. 设置 UAC 禁用
reg add
"HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Windows\CurrentVersion\Policies\Sys
tem" /v "LocalAccountTokenFilterPolicy" /t REG_DWORD /d "1" /f
# 以上语句设置 Windows 的 UAC 功能
```

```
rem=====
rem 6. 设置 Administrator(管理员) 及 Patrol(监控平台用户) 的密码
#net user administrator ???????? ???????? 请用实际密码代替
#net user patrol ***** ***** 请用实际密码代替
# 以上语句用于初始化重置 Administrator 和 Patrol 用户密码
```

```
rem=====
rem 7. 关闭计算机
#shutdown /t 3 /s (shutdown 前标注了 #, 所以此句只标注不执行)
:end
```

### 3.3.3 利用 Shell 脚本简化 Linux 逻辑节点的搬迁

下面是基于 Linux Shell 的脚本，主要实现功能为判断应用环境（研发和容灾），执行不同的 Shell 指令，将字典文件中的旧 IP 切换为对应的新 IP：

```
#!/bin/bash
cd /tmp/ipchange
#dos2unix ipchange.sh
#dos2unix ipcheck.txt

echo ++++++
echo + Welcome use redhat linux change patameter script +
echo ++++++

read -p "Disaster Recovery Environment(R)|Develop Testing
Environment(T)|Exit(E):" evar
# 以上语句显示欢迎页面，同时还显示应用环境菜单，根据操作的选取，可跳转到不同的环境
zb_ntp1=192.168.16.105 # (为 NTP1 服务器赋予指定 IP)
zb_ntp2=192.168.20.73 # (为 NTP2 服务器赋予指定 IP)
zb_dns1=192.168.16.105 # (为主 DNS 服务器赋予指定 IP)
zb_dns2=192.168.16.106 # (为辅 DNS 服务器赋予指定 IP)
zb_yum= # (初始化 YUM 服务器变量)
# 以上语句设置灾备环境的所有参数

kfcs_ntp1=192.168.127.103 # (为 NTP1 服务器赋予指定 IP)
kfcs_ntp2= # (初始化 NTP2 服务器变量)
kfcs_dns1=192.168.127.105 # (为主 DNS 服务器赋予指定 IP)
kfcs_dns2=192.168.127.106 # (为辅 DNS 服务器赋予指定 IP)
kfcs_yum= # (初始化 YUM 服务器变量)
# 以上语句设置开发环境的所有参数

dr(){ # (设置灾备环境的主程序)
#####
# Disaster Recovery Environment #
#####
# 保存旧 IP 信息
grep -i ipaddr /etc/sysconfig/network-scripts/ifcfg-eth0>/tmp/oldip.txt
awk -F= '{print $2}' /tmp/oldip.txt>/tmp/ipchange/oldip.txt
rm -f /tmp/oldip.txt
oldip=`awk '{print $0}' /tmp/ipchange/oldip.txt`
echo oldip=$oldip
echo

# 保存旧网关信息
grep -i gateway
```

```

/etc/sysconfig/network-scripts/ifcfg-eth0>/tmp/oldgatewayip.txt
awk -F="{" '{print $2}' /tmp/oldgatewayip.txt>/tmp/ipchange/oldgatewayip.txt
rm -f /tmp/oldgatewayip.txt
oldgatewayip=`awk '{print $0}' /tmp/ipchange/oldgatewayip.txt`
echo oldgatewayip=$oldgatewayip
echo

```

```

# 从字典文件中截取新旧 IP 对比值, 提取新 IP
grep "$oldip\>" /tmp/ipchange/ipcheck.txt>/tmp/oldtonewip.txt
awk '{print $2}' /tmp/oldtonewip.txt>/tmp/ipchange/newip.txt
mv /tmp/oldtonewip.txt /tmp/ipchange/oldtonewip.txt
newip=`awk '{print $1}' /tmp/ipchange/newip.txt`
echo newip=$newip
echo

```

```

# 在字典文件中进行比对, 如果找不到 IP 对应关系则退出
count=`grep "$oldip\>" /tmp/ipchange/ipcheck.txt |wc -l`
if [ $count -eq 0 ];then
echo "no ip matched in ipcheck.txt, will exit in 5 seconds!!!"
sleep 5
exit 100
fi
grep "$oldip\>" /tmp/ipchange/ipcheck.txt>/tmp/oldtonewip.txt
awk '{print $2}' /tmp/oldtonewip.txt>/tmp/ipchange/newip.txt
mv /tmp/oldtonewip.txt /tmp/ipchange/oldtonewip.txt
newip=`awk '{print $1}' /tmp/ipchange/newip.txt`
echo newip=$newip
echo

```

```

# 定义新网关
awk -F. '{print $1"."$2"."$3"."254}'
/tmp/ipchange/newip.txt>/tmp/ipchange/newgatewayip.txt
newgatewayip=`awk '{print $1}' /tmp/ipchange/newgatewayip.txt`
echo newgatewayip=$newgatewayip
echo

```

```

# 保存网卡的旧配置, 并设置新定义的配置信息
cd /etc/sysconfig/network-scripts/
cp ifcfg-eth0 /tmp/ipchange/ifcfg-eth0.old
sed -e "s/$oldip/$newip/g" ifcfg-eth0>ifcfg-eth0.tmp1
sed -e "s/$oldgatewayip/$newgatewayip/g" ifcfg-eth0.tmp1>ifcfg-eth0.tmp2
sed -e "/NETMASK/d" ifcfg-eth0.tmp2>ifcfg-eth0.tmp1
sed -e "/IPADDR/a\\NETMASK=255.255.255.0" ifcfg-eth0.tmp1>ifcfg-eth0.tmp2

rm -f ifcfg-eth0
cp ifcfg-eth0.tmp2 /tmp/ipchange/ifcfg-eth0.new
cp ifcfg-eth0.tmp2 ifcfg-eth0
rm -f ifcfg-eth0.tmp1
rm -f ifcfg-eth0.tmp2

```

```

cat /etc/sysconfig/network-scripts/ifcfg-eth0

```

```
#####
# 备份 hosts 文件
cp -f /etc/hosts /tmp/ipchange/oldhosts.txt

# 根据字典定义的文件, 替换 hosts 内关联节点对应的 IP
for ip in $(cat /etc/hosts | egrep ^[0-9] | awk '{print $1}' | grep -v 127.0.0.1)
do
iprule=$(grep $ip /tmp/ipchange/ipcheck.txt | head -n 1)
if [ "$iprule" != "" ]
then
shipsec=$(echo $iprule | awk '{print $1}' | awk -F'.' '{print $1"."$2"."$3}')
cdipsec=$(echo $iprule | awk '{print $2}' | awk -F'.' '{print $1"."$2"."$3}')
echo "shipsec:$shipsec"
echo "cdipsec:$cdipsec"
sed -i "s/^$shipsec/$cdipsec/g" /etc/hosts
else
echo "WARNING: ip $ip not matched in ipcheck.txt"
fi
done

#####
# 设置 NTP 信息 (通过 crontab 定时更新)
crontab -l >/tmp/crontab.old
if [ -f /var/spool/cron/root ]
then
sed -i '/ntpdate/d' /var/spool/cron/root
fi
echo "*/5 * * * * /sbin/ntpdate $zb_ntp1">>/var/spool/cron/root
echo "*/5 * * * * /sbin/ntpdate $zb_ntp2">>/var/spool/cron/root

echo "tinker panic 0">>/etc/ntp.conf
echo "restrict 127.0.0.1">>/etc/ntp.conf
echo "restrict default kod nomodify notrap">>/etc/ntp.conf
echo "server $zb_ntp1">>/etc/ntp.conf
echo "server $zb_ntp2">>/etc/ntp.conf
echo "keys /etc/ntp/keys">>/etc/ntp.conf
echo "driftfile /var/lib/ntp/drift">>/etc/ntp.conf

echo "$zb_ntp1">>/etc/ntp/ntpervers
echo "$zb_ntp2">>/etc/ntp/ntpervers
echo "clock.redhat.com">>/etc/ntp/ntpervers
echo "clock2.redhat.com">>/etc/ntp/ntpervers
echo "$zb_ntp1">>/etc/ntp/step-tickers
echo "$zb_ntp2">>/etc/ntp/step-tickers
echo "clock.redhat.com">>/etc/ntp/step-tickers
echo "clock2.redhat.com">>/etc/ntp/step-tickers

ntp=`cat /etc/ntp/ntpervers | head -1`
crontab -l
```

```
#####
# 设置 DNS 信息
cp /etc/resolv.conf /tmp/ipchange/resolv.conf.old
echo domain cpic.com.cn>/etc/resolv.conf
echo nameserver $zb_dns1>>/etc/resolv.conf
echo nameserver $zb_dns2>>/etc/resolv.conf

dns=`sed -e '/nameserver/g' /etc/resolv.conf | awk -F " " '{print $2}'
/etc/resolv.conf | head -2 | tail -1`

#####
# 设置监控平台参数
mv /home/patrol/startagent.sh /tmp/startagent.sh.old
echo su - patrol -c \" /home/patrol/Patrol3/PatrolAgent -id
$newip\">/home/patrol/startagent.sh
chown patrol:patrol /home/patrol/startagent.sh
chmod 774 /home/patrol/startagent.sh

#####
# 重置关键用户密码
#echo ???????? | passwd root --stdin          ???????? 请用实际密码代替
#echo ***** | passwd patrol --stdin          ***** 请用实际密码代替
#####
#shutdown computer
#shutdown -h now                                (shutdown 前标注了 #, 所以此句只标注不执行)
service network restart
service ntpd stop
chkconfig ntpd off
echo "New IP is $newip"
echo "New primary DNS is $dns"
echo "New primary NTP is $ntp"
echo "Change is successful!"
}
# 以上语句在设置完所有配置信息后, 需要重启网络服务, 以确保新 IP 能够正常工作。

dt(){                                           # (设置开发环境的主程序)
#####
# 保存旧 IP 信息
grep -i ipaddr /etc/sysconfig/network-scripts/ifcfg-eth0>/tmp/oldip.txt
awk -F= '{print $2}' /tmp/oldip.txt>/tmp/ipchange/oldip.txt
rm -f /tmp/oldip.txt
oldip=`awk '{print $0}' /tmp/ipchange/oldip.txt`
echo oldip=$oldip
echo

# 保存旧网关信息
grep -i gateway
/etc/sysconfig/network-scripts/ifcfg-eth0>/tmp/oldgatewayip.txt
awk -F= '{print $2}' /tmp/oldgatewayip.txt>/tmp/ipchange/oldgatewayip.txt
rm -f /tmp/oldgatewayip.txt
```

```

oldgatewayip=`awk '{print $0}' /tmp/ipchange/oldgatewayip.txt`
echo oldgatewayip=$oldgatewayip
echo

# 从字典文件中截取新旧 IP 对比值, 提取新 IP
grep "$oldip\>" /tmp/ipchange/ipcheck.txt>/tmp/oldtonewip.txt
awk '{print $2}' /tmp/oldtonewip.txt>/tmp/ipchange/newip.txt
mv /tmp/oldtonewip.txt /tmp/ipchange/oldtonewip.txt
newip=`awk '{print $1}' /tmp/ipchange/newip.txt`
echo newip=$newip
echo

# 在字典文件中比对, 如果找不到 IP 对应关系则退出
count=`grep "$oldip\>" /tmp/ipchange/ipcheck.txt |wc -l`
if [ $count -eq 0 ];then
echo "no ip matched in ipcheck.txt, will exit in 5 seconds!!!"
sleep 5
exit 100
fi
grep "$oldip\>" /tmp/ipchange/ipcheck.txt>/tmp/oldtonewip.txt
awk '{print $2}' /tmp/oldtonewip.txt>/tmp/ipchange/newip.txt
mv /tmp/oldtonewip.txt /tmp/ipchange/oldtonewip.txt
newip=`awk '{print $1}' /tmp/ipchange/newip.txt`
echo newip=$newip
echo

# 定义新网关
awk -F. '{print $1"."$2"."$3"."254}'
/tmp/ipchange/newip.txt>/tmp/ipchange/newgatewayip.txt
newgatewayip=`awk '{print $1}' /tmp/ipchange/newgatewayip.txt`
echo newgatewayip=$newgatewayip
echo

# 保存网卡的旧配置, 并设置新定义的配置信息
cd /etc/sysconfig/network-scripts/
cp ifcfg-eth0 /tmp/ipchange/ifcfg-eth0.old
sed -e "s/$oldip/$newip/g" ifcfg-eth0>ifcfg-eth0.tmp1
sed -e "s/$oldgatewayip/$newgatewayip/g" ifcfg-eth0.tmp1>ifcfg-eth0.tmp2
sed -e "/NETMASK/d" ifcfg-eth0.tmp2>ifcfg-eth0.tmp1
sed -e "/IPADDR/a\\NETMASK=255.255.255.0" ifcfg-eth0.tmp1>ifcfg-eth0.tmp2

rm -f ifcfg-eth0
cp ifcfg-eth0.tmp2 /tmp/ipchange/ifcfg-eth0.new
cp ifcfg-eth0.tmp2 ifcfg-eth0
rm -f ifcfg-eth0.tmp1
rm -f ifcfg-eth0.tmp2

cat /etc/sysconfig/network-scripts/ifcfg-eth0
#####
# 备份 hosts 文件
cp -f /etc/hosts /tmp/ipchange/oldhosts.txt

```



```

for ip in $(cat /etc/hosts | egrep ^[0-9] | awk '{print $1}' | grep -v 127.0.0.1)
do
    iprule=$(grep $ip /tmp/ipchange/ipcheck.txt | head -n 1)
    if [ "$iprule" != "" ]
    then
        shipsec=$(echo $iprule | awk '{print $1}' | awk -F'.' '{print $1"."$2"."$3}')
        cdipsec=$(echo $iprule | awk '{print $2}' | awk -F'.' '{print $1"."$2"."$3}')
        echo "shipsec:$shipsec"
        echo "cdipsec:$cdipsec"
        sed -i "s/^\$shipsec/\$cdipsec/g" /etc/hosts
    else
        echo "WARNING: ip $ip not matched in ipcheck.txt"
    fi
done

#####
# 设置 NTP 信息 (通过 crontab 定时更新)
crontab -l >/tmp/crontab.old
if [ -f /var/spool/cron/root ]
then
    sed -i '/ntpdate/d' /var/spool/cron/root
fi
echo "*/5 * * * * /sbin/ntpdate $kfcs_ntp1">>/var/spool/cron/root
#echo "*/5 * * * * /sbin/ntpdate $kfcs_ntp2">>/var/spool/cron/root

echo "tinker panic 0">/etc/ntp.conf
echo "restrict 127.0.0.1">>/etc/ntp.conf
echo "restrict default kod nomodify notrap">>/etc/ntp.conf
echo "server $kfcs_ntp1">>/etc/ntp.conf
echo "server $kfcs_ntp2">>/etc/ntp.conf
echo "keys /etc/ntp/keys">>/etc/ntp.conf
echo "driftfile /var/lib/ntp/drift">>/etc/ntp.conf

echo "$kfcs_ntp1">/etc/ntp/ntpervers
echo "$kfcs_ntp2">>/etc/ntp/ntpervers
echo "clock.redhat.com">>/etc/ntp/ntpervers
echo "clock2.redhat.com">>/etc/ntp/ntpervers
echo "$kfcs_ntp1">/etc/ntp/step-tickers
echo "$kfcs_ntp2">>/etc/ntp/step-tickers
echo "clock.redhat.com">>/etc/ntp/step-tickers
echo "clock2.redhat.com">>/etc/ntp/step-tickers

ntp=`cat /etc/ntp/ntpervers | head -1`

crontab -l
#####
# 设置 DNS 信息
cp /etc/resolv.conf /tmp/ipchange/resolv.conf.old
> /etc/resolv.conf
dns=`sed -e '/nameserver/g' /etc/resolv.conf | awk -F " " '{print $2}' /etc/`

```

```

resolv.conf | head -2 | tail -1`

#####
# 设置监控平台参数
mv /home/patrol/startagent.sh /tmp/startagent.sh.old
echo su - patrol -c \" /home/patrol/Patrol3/PatrolAgent -id
$newip\">/home/patrol/startagent.sh
chown patrol:patrol /home/patrol/startagent.sh
chmod 774 /home/patrol/startagent.sh

#####
# 重置关键用户密码
#echo ??????? | passwd root --stdin          ??????? 请用实际密码代替
#echo ??????? | passwd patrol --stdin         ??????? 请用实际密码代替
#####
#shutdown computer
#shutdown -h now                               (shutdown 前标注了 #, 所以此句只标注不执行)
service network restart
service ntpd stop
chkconfig ntpd off
echo "New IP is $newip"
echo "New primary DNS is $dns"
echo "New primary NTP is $ntp"
echo "Change is successful!"
}
# 以上语句在设置完所有配置信息后, 需要重启网络服务, 以确保新 IP 能够正常工作。

ex() {
echo "Nothing is changed!"
exit 0
}
# 以上语句为设置 CASE 语句场景的 ex 函数, 选取 E|e 时, 不执行任何操作, 直接退出。
case "$sevar" in
R|r)
dr
;;
T|t)
dt
;;
E|e)
ex
;;
*)
# echo $"Usage: $0 {R|T|E}"
# exit 1
Esac
# 以上语句为 CASE 语句场景, 可根据不同的选择执行灾备环境、开发环境的参数调整指令,
或者不执行任何操作, 直接退出。

```

以下为 Linux 示例的截图:

原始状态, 打开虚拟机后初始 IP 的后两位为 192.101, 如图 3-21 所示。

```
File Edit View Terminal Tabs Help
[root@JTSXDXAPP1-DR/tmp/ipchange]#ifconfig
eth0      Link encap:Ethernet  HWaddr 00:50:56:84:04:9C
          inet addr:192.101.101.101  Bcast:192.255.255.255  Mask:255.255.255.0
          inet6 addr: fe80::250:56ff:fe84:49c/64 Scope:Link
          UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
          RX packets:182 errors:0 dropped:0 overruns:0 frame:0
          TX packets:126 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:1000
          RX bytes:12054 (11.7 KiB)  TX bytes:11416 (11.1 KiB)
          Interrupt:59 Base address:0x2000
```

图 3-21 旧 IP 显示截图

执行 ipchange 脚本时，要求用户根据提示选取执行环境，我们输入 R（灾备环境），如图 3-22 所示。

```
File Edit View Terminal Tabs Help
[root@JTSXDXAPP1-DR/tmp/ipchange]#ls
ipchange.sh  ipcheck.txt
[root@JTSXDXAPP1-DR/tmp/ipchange]#./ipchange.sh
+++++
+ Welcome use redhat linux change patameter script +
+++++
Disaster Recovery Environment(R)|Develop Testing Environment(T)|Exit(E):
```

图 3-22 选取执行环境截图

执行 ipchange 脚本后，我们可以看到切换成功的提示，如图 3-23 所示。

```
cdipsec: 16
shipsec: 16
cdipsec: 16
shipsec: 16
cdipsec: 16
shipsec: 16
cdipsec: 16
shipsec: 16
cdipsec: 16
shipsec: 177
cdipsec: 20
00 1 * * * /app/autorun/cleareclientqueue.sh
*/5 * * * ntpdate 10.101.101.105
*/5 * * * ntpdate 10.101.101.105
Shutting down interface eth0: [ OK ]
Shutting down loopback interface: [ OK ]
Bringing up loopback interface: [ OK ]
Bringing up interface eth0: [ OK ]
Shutting down ntpd: [ FAILED ]
New primary DNS is 105
New primary NTP is 105
Change is successful
[root@JTSXDXAPP1-DR/tmp/ipchange]#
```

图 3-23 显示 IP 替换完成

执行 ipchange 脚本后，可以看到最后两位已经替换为 100.101 了，如图 3-24 所示。

```
File Edit View Terminal Tabs Help
[root@JTSXDXAPP1-DR]#ifconfig
eth0      Link encap:Ethernet  HWaddr 00:50:56:84:04:9C
          inet addr:100.101.101.101  Bcast:100.255.255.255  Mask:255.255.255.0
          inet6 addr: fe80::250:56ff:fe84:49c/64 Scope:Link
          UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
          RX packets:398 errors:0 dropped:0 overruns:0 frame:0
          TX packets:258 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:1000
          RX bytes:25732 (25.1 KiB)  TX bytes:23580 (23.0 KiB)
```

图 3-24 新 IP 显示截图

### 3.3.4 通过 SFTP 和 WMIC 指令将脚本文件上传至所有虚拟机

3.3.2 和 3.3.3 节我们完成了切换脚本的编制，我们要在源存储与目标存储保持复制关系的期间，将切换脚本及字典文件上传至所有逻辑节点的指定目录，以确保复制关系断开后目标虚拟机中存在切换用的脚本。

#### 1. 基于 Linux 的上传

所有 Linux 默认都通过 SSH 方式访问，因此我们可以利用 SFTP 功能模块将切换脚本和字典文件上传至所有 Linux 节点的指定位置。

(1) 先搭建一台上传用的 Linux 脚本机，安装 LFTP 包，将涉及 Linux 节点的 IP 保存为 /tmp/ip.txt。

(2) 同时编制以下脚本：

```
#!/bin/bash
for i in `cat /tmp/ip.txt`
do
lftp -u 用户名@密码 sftp://$i <<EOF >>/tmp/sftp.log
mkdir /tmp/ipchange
cd /tmp/ipchange
lcd /tmp
mput ipchange.sh ipcheck.txt
EOF
done
```

#### 2. 基于 Windows 的上传

环境中所有 Windows 节点均启用了 WMI 管理模块，WMIC 提供了从命令行接口和批命令脚本执行系统管理的支持。因此我们可以利用 WMIC 将切换脚本和字典文件上传至所有 Windows 节点的指定位置。

我们同样首先搭建一台上传用的 Windows 脚本机，将涉及 Windows 节点的 IP 保存为 d:\windows-script\ip.txt，同时编制以下脚本：

```
@echo off
del d:\windows-script\ip.log
for /f "skip=1 eol=# tokens=1,2,3 delims= " %i in (d:\windows-script\ip.txt)
do (@echo %i >> d:\windows-script\ip.log
net use s: \\%i\c$ /user:%j %k >> d:\windows-script\ip.log
mkdir s:\ipchange
xcopy d:\windows-script\ipchange.cmd s:\ipchange /s /y >>
d:\windows-script\ip.log
xcopy d:\windows-script\ipcheck.txt s:\ipchange /s /y >>
d:\windows-script\ip.log
net use s: /delete >> ip.log
)
```

### 3.3.5 搬迁期间的注意事项

由于是通过存储底层复制技术进行的逻辑迁移，因此在目标端完成注册，并调整虚拟机名称后，在 vCenter 管理界面看到的是改名为目标 IP 的虚拟机名称。但存储底层的数据文件仍然是以原始 IP 命名的。

为防止给以后的运维留下后患，需要对虚拟机进行一次存储 VMotion 的操作，这样存储内的虚拟机文件名在存储迁移的同时就能自动调整为新 IP 的文件名了。

浏览存储底层所看到的文件情况，如图 3-25 所示。

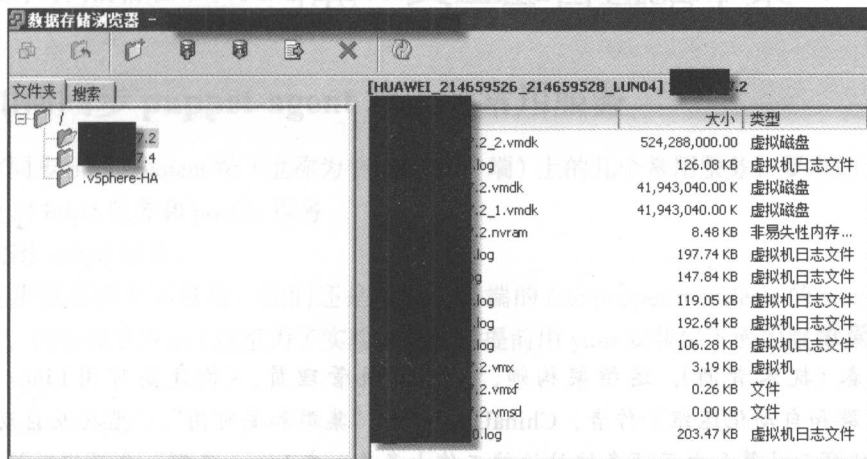


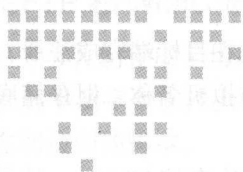
图 3-25 数据存储内部虚拟机的文件结构

以上的脚本是利用 Shell 脚本简化 Linux 逻辑节点的搬迁，在实际运行中还有不完善的地方。由于未包含操作系统判别语句，目前仅在 RHEL5 和 RHEL6 系列上测试通过，因此大家可以根据实际情况添加判断语句以适应 RHEL 较新的 7 版本或其他 Linux 版本。另外也可完善逻辑语句使得脚本更加严谨可靠。

字典文件 ipcheck.txt 中的 IP 对应关系，需要确保 IP 地址的唯一，相同的新旧 IP 对应关系只能有一行。同时也要防止同一个旧 IP 对应多个新 IP 的情况。

## 3.4 小结

作为一名 IT 运维人员，在日常的运维中，只要熟练掌握基本工具并融会贯通加以灵活应用，就可以将看似不可能完成的任务变为可能，同时达到事半功倍的效果。以上经验仅供大家分享。在此感谢李彬厚和齐特两位同事对我的大力支持。



## 集中配置管理工具 Puppet

### 作者简介

余洪春(抚琴煮酒), 运维架构师、资深系统管理员,《构建高可用 Linux 服务器》《Linux 集群和自动化运维》作者, ChinaUnix 论坛“集群和高可用”、“监控及自动化运维”版版主。从事云计算和电子商务网站运维工作十多年, 在 Linux 集群、自动化运维、云计算及高并发高流量网站架构设计等方面进行了深入的研究, 在大量一线实践中积累了丰富的经验。精通负载均衡高可用和自动化运维技术, 擅长高并发高流量网站系统架构设计。

大数据时代高伸缩性、容错性、分布式的特点给系统运维提出了更高的要求, 现在已不再是系统管理员疲于安装操作系统、对系统参数进行逐一配置与优化、打补丁、安装软件、配置软件、添加某个服务的时代。为了提高效率、避免重复劳动、减少错误、积累知识, 系统管理员已经开始做一些局部的自动化工作。但只有这些还远远不够, 为了满足运维需求, 还需要更彻底地应用自动化运维工具。

常见的运维工作流程包括: 安装系统→优化系统与配置→安装软件→配置软件→添加监控→检查, 等网站或系统正式上线后, 后续可能还会有添加服务→配置变更→打补丁修复漏洞等工作, 是不是觉得很繁琐? 尤其当我们负责部署和维护大量服务器, 凭借一己之力无法完成时, 便需要一些自动化工具来帮忙了。

系统运维工作面临的各种不确定性更让人头疼, 在 10 台机器上变更应用还是很简单的事, 就算到了 100 台也有轻量级的 Fabric 自动化运维工具, 但是如果上升到 1000 台甚至上万台时就会变得非常复杂。重复性的劳动还会让人觉得疲惫和乏味, 久而久之可能还会产生厌倦工作的情绪, 这个时候使用集中配置管理工具 Puppet 则能很轻松地解决这些问题, 这也



是现在 Puppet 越来越流行的原因之一。

下面将介绍 Puppet 的进阶操作。

这里的操作环境按照前面的操作已经搭建完毕，具体如下所示：

```
server.cn7788.com 192.168.1.205 puppet-master
fabric.cn7788.com 192.168.1.204 puppet-client
client.cn7788.com 192.168.1.206 puppet-client
```

我们清理好前面的环境，清空 /etc/puppet/manifests/ 里的 site.pp 文件的内容，并且利用 ntpdate 做好时间的精准对时，不然 puppet-client 在连接时会报错，连接不上。

## 4.1 如何同步 puppet-agent 端上的常用服务

如何同步 puppet-agent 端（也称为 Puppet 客户端）上的几个常用服务，要求如下：

□ 开启 httpd 服务和 postfix 服务。

□ 关闭 vsftpd 服务。

实现步骤其实并不复杂，我们还是在服务器端的 /etc/puppet/manifests 的 site.pp 文件上进行操作，内容如下所示（这里为了实验方便，请提前用 yum 安装好下面的服务）：

```
service {
  ["httpd", "postfix"]:
  ensure => running;
  "vsftpd":
  ensure => stopped;
}
```

Puppet 客户端顺利连接上服务器端后，正常显示结果应该如下所示（完成需求，以节点机器 client.cn7788.com 举例说明）：

```
Info: Retrieving pluginfacts
Info: Retrieving plugin
Info: Caching catalog for client.cn7788.com
Info: Applying configuration version '1446608434'
Notice: /Stage[main]/Main/Service[postfix]/ensure: ensure changed 'stopped' to 'running'
Info: /Stage[main]/Main/Service[postfix]: Unscheduling refresh on Service[postfix]
Notice: Finished catalog run in 1.60 seconds
```

我们可在 Puppet 客户端下面输入命令来验证下实验结果，具体命令如下所示：

```
service postfix status
master (pid 10028) is running...
service httpd status
httpd (pid 9603) is running...
```

```
service vsftpd status
vsftpd is stopped
```

## 4.2 如何在 puppet-agent 端上自动安装常用的软件包

在 Puppet 客户端自动安装 screen、ntp 和 sysstat 包，具体步骤如下所示。

(1) 编辑 /etc/puppet/manifests/site.pp 文件，内容如下：

```
package {
  ["screen", "ntp", "sysstat"]:
  ensure => "installed";
}
```

(2) 客户端顺利连接上服务器端后，正常显示结果应该如下所示（完成需求，以节点机器 client.cn7788.com 举例说明）：

```
Info: Retrieving pluginfacts
Info: Retrieving plugin
Info: Caching catalog for client.cn7788.com
Info: Applying configuration version '1446608680'
Notice: /Stage[main]/Main/Package[sysstat]/ensure: created
Notice: Finished catalog run in 15.03 seconds
```

## 4.3 如何自动同步 puppet-agent 端的 yum 源

通过以下命令我们可以观察到 puppet-master 端机器上更新了 yum 源：

```
ls -ll /etc/yum.repos.d/
```

命令显示结果如下所示：

```
total 24
-rw-r--r--. 1 root root 1926 Feb 25 2013 CentOS-Base.repo
-rw-r--r--. 1 root root 638 Feb 25 2013 CentOS-Debuginfo.repo
-rw-r--r--. 1 root root 630 Feb 25 2013 CentOS-Media.repo
-rw-r--r--. 1 root root 3664 Feb 25 2013 CentOS-Vault.repo
-rw-r--r--. 1 root root 957 Nov 4 03:46 epel.repo
-rw-r--r--. 1 root root 0 Nov 4 03:46 nginx.repo
-rw-r--r--. 1 root root 1250 Apr 12 2013 puppetlabs.repo
```

如果想通过 Puppet 将这些文件都推送到与 client.cn7788.com 和 fabric.cn7788.com 相对应的目录 /etc/yum.repos.d/ 下面去，那么这个该如何实现呢？

(1) 修改 /etc/puppet/fileservers.conf 文件，内容如下所示：

```
[files]
path /etc/yum.repos.d/
```

```
allow *
```

(2) 修改 `/etc/puppet/manifests/site.pp` 文件，内容如下所示：

```
file
{
  "/etc/yum.repos.d/":
  source => "puppet://server.cn7788.com/files/",
  group => root,
  owner => root,
  mode => 644,
  recurse => true,
  force => true,
  purge => true
}
```

Puppet 客户端正确连接到 Puppet 服务器端后，大家会发现，puppet-server 会向 puppet-client 端推送文件，我们分别在两台节点机器上输入如下命令：

```
puppet agent --test --server server.cn7788.com
```

两台机器的结果是一样的，输入命令的反馈结果如下所示：

```
Info: Retrieving pluginfacts
Info: Retrieving plugin
Info: Caching catalog for client.cn7788.com
Info: Applying configuration version '1446619419'
Notice: /Stage[main]/Main/File[/etc/yum.repos.d/puppetlabs.repo]/ensure:
defined content as '{md5}14d68f86efb69e928d626c7ea8a974b3'
Notice: /Stage[main]/Main/File[/etc/yum.repos.d/CentOS-Debuginfo.repo]/ensure:
defined content as '{md5}8b95e819eeea42849932309b5be5533d'
Info: Computing checksum on file /etc/yum.repos.d/puppetlabs-release-6-7.
noarch.rpm
Info: /Stage[main]/Main/File[/etc/yum.repos.d/puppetlabs-release-6-7.noarch.
rpm]: Filebucketed /etc/yum.repos.d/puppetlabs-release-6-7.noarch.rpm to puppet
with sum 2aa0affe57ade441bfb9dcd6126a6cd6
Notice: /Stage[main]/Main/File[/etc/yum.repos.d/puppetlabs-release-6-7.noarch.
rpm]/ensure: removed
Notice: /Stage[main]/Main/File[/etc/yum.repos.d/CentOS-Base.repo]/ensure:
defined content as '{md5}d03052aaa85e5d26451f0ada9054f50f'
Info: Computing checksum on file /etc/yum.repos.d/softlist
Info: /Stage[main]/Main/File[/etc/yum.repos.d/softlist]: Filebucketed /etc/yum.
repos.d/softlist to puppet with sum 36fe51e7e690fe7d65046e9868ed2fa4
Notice: /Stage[main]/Main/File[/etc/yum.repos.d/softlist]/ensure: removed
Info: /Stage[main]/Main/File[/etc/yum.repos.d/test]: Recursively backing up to
filebucket
Info: Computing checksum on file /etc/yum.repos.d/test/dd
Info: /Stage[main]/Main/File[/etc/yum.repos.d/test]: Filebucketed /etc/yum.
repos.d/test/dd to puppet with sum d41d8cd98f00b204e9800998ecf8427e
Info: Computing checksum on file /etc/yum.repos.d/test/cc
Info: FileBucket got a duplicate file {md5}d41d8cd98f00b204e9800998ecf8427e
Info: /Stage[main]/Main/File[/etc/yum.repos.d/test]: Filebucketed /etc/yum.
```

```

repos.d/test/cc to puppet with sum d41d8cd98f00b204e9800998ecf8427e
Notice: /Stage[main]/Main/File[/etc/yum.repos.d/test]/ensure: removed
Notice: /Stage[main]/Main/File[/etc/yum.repos.d/nginx.repo]/ensure: defined
content as '{md5}d41d8cd98f00b204e9800998ecf8427e'
Notice: /Stage[main]/Main/File[/etc/yum.repos.d/CentOS-Media.repo]/ensure:
defined content as '{md5}db3010a594efc3043651d78741ac02ff'
Notice: /Stage[main]/Main/File[/etc/yum.repos.d/epel.repo]/ensure: defined
content as '{md5}e8950030d9c72cf3e7a6469e8c1404ca'
Notice: /Stage[main]/Main/File[/etc/yum.repos.d/CentOS-Vault.repo]/ensure:
defined content as '{md5}62b394965682a15877a36357fe55689d'
Notice: Finished catalog run in 0.75 seconds

```

同样，我们在 Puppet 客户端用 `ll` 命令进行观察，将会发现已经将 puppet-server 端的 `/etc/yum.repos.d` 目录下的文件同步过来了，完成此工作需求，检查结果如下：

```

total 24
-rw-r--r--. 1 root root 1926 Nov  4 06:44 CentOS-Base.repo
-rw-r--r--. 1 root root  638 Nov  4 06:44 CentOS-Debuginfo.repo
-rw-r--r--. 1 root root  630 Nov  4 06:44 CentOS-Media.repo
-rw-r--r--. 1 root root 3664 Nov  4 06:44 CentOS-Vault.repo
-rw-r--r--. 1 root root  957 Nov  4 06:44 epel.repo
-rw-r--r--. 1 root root    0 Nov  4 06:44 nginx.repo
-rw-r--r--. 1 root root 1250 Nov  4 06:44 puppetlabs.repo

```

## 4.4 如何根据不同名字的节点机器推送不同的文件

Puppet 服务器端向名为 `client.cn7788.com` 的客户端推送 `/etc/crontab` 文件，向名为 `fabric.cn7788.com` 的客户端推送 `/etc/hosts` 文件，向名为 `nginx.cn7788.com` 的客户端推送 `/etc/resolv.conf` 文件，这个应该如何实现呢？

这里的需求比较复杂，我们可以通过 Puppet 模块来实现需求，模块文件一般是存放在服务器端的 `/etc/puppet/modules/` 下，在它下面我们会定义一个名为 `pushfile` 的模块。模块是 Puppet 生态系统中的核心部分，我们一般通过资源的定义告诉 Puppet 应该做什么，一般一个应用就编写成一个模块，例如我们定义的 `pushfile` 模块。

我们用 `#` 注释掉 `fileservers.conf` 文件里面的相关内容，清空 `site.pp` 文件，然后在 `/etc/puppet/modules` 下面建立名为 `pushfile` 的模块，操作如下所示：

```
mkdir -p /etc/puppet/modules/pushfile/{manifests,files,templates}
```

而 `site.pp` 文件的内容如下：

```
import "node.pp"
```

这里扩展了 `site.pp` 文件的内容，它会载入 `node.pp` 文件，这样 puppet-master 在启动的时候，就会自动接入并处理 `node.pp` 文件了。

服务器端的 `/etc/puppet/manifests/node.pp` 文件内容如下所示：

```
node 'client.cn7788.com'{
  file
  {"/etc/crontab":
    source => "puppet://server.cn7788.com/modules/pushfile/crontab",
    group => root,
    owner => root,
    mode => 644,
  }
}

node 'fabric.cn7788.com'{
  file
  {"/etc/hosts":
    source => "puppet://server.cn7788.com/modules/pushfile/hosts",
    group => root,
    owner => root,
    mode => 644,
  }
}

node 'nginx.cn7788.com'{
  file
  {"/etc/resolv.conf":
    source => "puppet://server.cn7788.com/modules/pushfile/resolv.conf",
    group => root,
    owner => root,
    mode => 644,
  }
}
```

`node.pp` 配置文件比较长并且也很复杂，我们究竟应该使用什么方法呢？我们可以输入如下命令：

```
puppet parser validate node.pp
```

如果配置文件是正确的，则什么也不显示；如果检测到是错误的，则会以红色醒目字体来提示。

Puppet 利用 `node`（节点）来区分不同的客户端，并且会给不同的客户端分配不同的资源，我们观察下 `node.pp` 文件，在这个文件里，`source` 值会告诉 Puppet 去哪里寻找文件，我们将文件都置于 puppet-server 的 `/etc/puppet/modules/site/files` 目录下面，它相当于根目录，然后将服务器端的文件依次复制到此目录的 `/etc` 下，操作如下所示：

```
cp /etc/{crontab,hosts,resolv.conf} /etc/puppet/modules/pushfile/files
```

我们用 `tree` 命令来查看下 `pushfile` 模块的目录树结构，命令如下：

```
tree /etc/puppet/modules/pushfile
```

命令显示结果如下所示：

```
/etc/puppet/modules/pushfile
├── files
│   ├── crontab
│   ├── hosts
│   └── resolv.conf
├── manifests
└── templates
3 directories, 3 files
```

我们依次在 3 台 puppet-client 机器上面执行 puppet 命令，可以发现配置是成功的，这里以 client.cn7788.com 机器举例说明下，输入如下命令：

```
puppet agent --test --server server.cn7788.com
```

命令显示结果如下所示：

```
Info: Retrieving pluginfacts
Info: Retrieving plugin
Info: Caching catalog for client.cn7788.com
Info: Applying configuration version '1446623887'
Notice: /Stage[main]/Main/Node[client.cn7788.com]/File[/etc/crontab]/content:
--- /etc/crontab 2015-11-03 07:34:26.372044003 +0000
+++ /tmp/puppet-file20151104-17989-alqkh6-02015-11-04 08:02:50.127072946 +0000
@@ -14,10 +14,9 @@
# | | | | |
# * * * * * user-name command to be executed

-00 01 * * * root /bin/bash /usr/local/nginx/sbin/cut_nginx_log.sh >> /dev/
null 2>&1
+#00 01 * * * root /bin/bash /usr/local/nginx/sbin/cut_nginx_log.sh >> /dev/
null 2>&1
01 02 * * * root /bin/bash /root/backup.sh >> /dev/null 2>&1

-03 03 * * * root /bin/bash /root/sshdeny.sh >> /dev/null 2>&1
-#01 04 * * * root /bin/bash /root/rsync_dir.sh >> /dev/null 2>&1
+#03 03 * * * root /bin/bash /root/sshdeny.sh >> /dev/null 2>&1

-##/5 * * * * root /etc/init.d/iptables stop
+*/5 * * * * root /etc/init.d/iptables stop

Info: Computing checksum on file /etc/crontab
Info: /Stage[main]/Main/Node[client.cn7788.com]/File[/etc/crontab]:
Filebucketed /etc/crontab to puppet with sum 7e76ef490e02dde0dd8e82a5cf7c0c69
Notice: /Stage[main]/Main/Node[client.cn7788.com]/File[/etc/crontab]/content:
content changed '{md5}7e76ef490e02dde0dd8e82a5cf7c0c69' to '{md5}2022061d798f39
33e0caafb614272212'
Notice: Finished catalog run in 0.47 seconds
```

配置是成功的，`/etc/crontab` 被成功地推送过来了，而且内容也进行了置换，其他节点机器的结果这里就不打印了。



**注意** 在 `/etc/puppet/modules` 下定义的模块都是自动载入的，所以不需要用 `import` 来加载。

## 4.5 如何根据节点机器名来选择性地执行 Shell 程序

客户端机器 `nginx.cn7788.com` 没有安装 Nagios 客户端程序，这时想通过 `puppet-server` 推送 Shell 脚本自动安装，其他节点机器暂时不需要安装这个程序，那么这个应该如何实现呢？

我们主要还是通过模块的方法来实现这个需求，和 4.4 节一样，先在这里建立一个名为 `nagioscli` 的模块，命令如下所示：

```
mkdir -p /etc/puppet/modules/nagiosins/{manifests,files,templates}
```

在 `/etc/puppet/modules/nagioscli/files` 目录下安装 Nagios 客户端名为 `nagioscli.sh` 的 Shell 程序，具体内容如下：

```
#!/bin/bash
useradd nagios
cd /usr/local/src
wget wget http://syslab.comsenz.com/downloads/linux/nagios-plugins-1.4.13.tar.gz
wget http://syslab.comsenz.com/downloads/linux/nrpe-2.12.tar.gz
tar zxvf nagios-plugins-1.4.13.tar.gz
cd nagios-plugins-1.4.13
./configure
make
make install
chown nagios:nagios /usr/local/nagios
chown -R nagios:nagios /usr/local/nagios/libexec
cd ../
tar zxvf nrpe-2.12.tar.gz
cd nrpe-2.12
./configure
make all
make install-plugin
make install-daemon
make install-daemon-config
sed -i 's@allowed_hosts=127.0.0.1@allowed_hosts=114.112.11.11@' /usr/local/
nagios/etc/nrpe.cfg
/usr/local/nagios/bin/nrpe -c /usr/local/nagios/etc/nrpe.cfg -d
echo "/usr/local/nagios/bin/nrpe -c /usr/local/nagios/etc/nrpe.cfg -d" >> /etc/
rc.local
```

`node.pp` 文件的内容如下所示：

```
node 'nginx.cn7788.com'{
```



```

file
{"/usr/local/src/nagiosins.sh":
source => "puppet://server.cn7788.com/modules/nagiosins/nagiosins.sh",
group => root,
owner => root,
mode => 755,
}

exec {
"auto install naigios client":
command => "sh /usr/local/src/nagiosins.sh",
user => "root",
path => ["/usr/bin", "/usr/sbin", "/bin", "/bin/sh"],
}

node 'client.cn7788.com'{
file
{"/usr/local/src/nagiosins.sh":
source => "puppet://server.cn7788.com/modules/nagiosins/nagiosins.sh",
group => root,
owner => root,
mode => 755,
}

exec {
"auto install naigios client":
command => "sh /usr/local/src/nagiosins.sh",
user => "root",
path => ["/usr/bin", "/usr/sbin", "/bin", "/bin/sh"],
}
}

node 'fabric.cn7788.com'{
}

```

节点机器 client 和 fabric 机器后面什么都没有，表示在此节点机器上面没有任何操作，因为 client 和 fabric 节点机器也在此 Puppet 环境里，并配置成了自动连接。如此配置，是为了防止自动连接时 Puppet 频繁报错。

这里以 client.cn7788.com 为例，在其主机上输入如下命令：

```
puppetd --test --server server.cn7788.com
```

命令显示结果如下所示：

```

Info: Retrieving pluginfacts
Info: Retrieving plugin
Info: Caching catalog for client.cn7788.com
Info: Applying configuration version '1446693418'

```

```
Notice: /Stage[main]/Main/Node[client.cn7788.com]/Exec[auto install nagios
client]/returns: executed successfully
Notice: Finished catalog run in 165.27 seconds
```

执行时间比较长，总共耗时 165.27 秒，我们先检查下 client.cn7788.com 的节点机器上是否开启了 nrpe 进程，输入命令如下所示：

```
ps aux | grep nrpe | grep -v grep
```

命令显示结果如下所示：

```
nagios 22331 0.0 0.1 5108 924 ? Ss 22:35 0:00 /usr/local/
nagios/bin/nrpe -c /usr/local/nagios/etc/nrpe.cfg -d
```

我们再检查下 /etc/rc.local，看看此命令有没有添加进去，命令如下所示：

```
grep -v "^#" /etc/rc.local
```

命令执行结果显示如下所示：

```
touch /var/lock/subsys/local
/usr/local/nagios/bin/nrpe -c /usr/local/nagios/etc/nrpe.cfg -d
```

检查结果说明 puppet-master 的 nagioscli 模块是正常的，nginx.cn7788.com 的结果与此类似，这里就不贴出检测结果了。

## 4.6 如何快速同步 puppet-server 端的 www 目录文件

当我们拥有大规模 Web 集群的时候，所有服务器 /var/www/html 的数据要求必须迅速统一，这个又该如何实现呢？

实例说明：server.cn7788.com 机器下的 /data/svn/resource 目录文件或子目录发生改变时，要求 nginx.cn7788.com、client.cn7788.com 和 fabric.cn7788.com 对应的目录 /var/www/html 文件或子目录也发生改变。

前面已经提到了，Puppet 分发大文件和海量图片小文件的效果并不好，但这里我们其实可以用 rsync+Puppet 的方式来实现工作中的需求。这里要用到 Puppet Kick 的知识点，Puppet Kick 是 puppet-master 端使用此命令强制其 Agent 节点机器运行 Puppet Agent，从而达到立即更新或同步文件的目的。当然也可以用 Puppet rsync 模块，但我个人觉得使用这个太麻烦了，所以这里还是采用自己摸索出来的方法，具体步骤如下（这里以 client.cn7788.com 节点机器举例说明，其他 Puppet 客户端操作与此类似，这里就不一一列举了）。

（1）在所有 puppet-client 客户端上配置 puppet.conf 文件，使其固定使用 8139 端口，我们在其 /etc/puppet/puppet.conf 文件下添加如下命令：

```
[puppet-client]
```

```
listen = true
server=server.cn7788.com
```

listen=true: 此选项将使 puppet agent 监听 8139 端口。

server=server.cn7788.com: 此选项必须要配置, 经过测试, 可以发现如果无此选项时, puppet-client 会连接不到 puppet-master 机器, 从而导致文件同步不过去。

(2) 修改客户端的 /etc/puppet/auth.conf, 允许 server.cn7788.com 的服务器端进行推送。

在最末行的 path/ 之后添加 allow\*, 保证代码内容相同:

```
path /run
auth any
allow *
```

如果不进行此项操作的话, 会有如下报错:

```
Debug: /File[/var/lib/puppet/ssl/private]: Autorequiring File[/var/lib/puppet/ssl]
Debug: /File[/var/lib/puppet/ssl/certs/server.cn7788.com.pem]: Autorequiring
File[/var/lib/puppet/ssl/certs]
Debug: /File[/var/lib/puppet/ssl/private_keys]: Autorequiring File[/var/lib/
puppet/ssl]
Debug: /File[/var/lib/puppet/lib]: Autorequiring File[/var/lib/puppet]
Debug: /File[/var/lib/puppet/ssl/private_keys/server.cn7788.com.pem]:
Autorequiring File[/var/lib/puppet/ssl/private_keys]
Debug: /File[/var/lib/puppet/ssl/certificate_requests]: Autorequiring File[/
var/lib/puppet/ssl]
Debug: /File[/var/lib/puppet/state]: Autorequiring File[/var/lib/puppet]
Debug: /File[/var/lib/puppet/ssl]: Autorequiring File[/var/lib/puppet]
Debug: /File[/var/lib/puppet/ssl/crl.pem]: Autorequiring File[/var/lib/puppet/ssl]
Debug: /File[/var/lib/puppet/facts.d]: Autorequiring File[/var/lib/puppet]
Debug: /File[/var/lib/puppet/ssl/certs]: Autorequiring File[/var/lib/puppet/ssl]
Debug: /File[/var/lib/puppet/ssl/public_keys]: Autorequiring File[/var/lib/
puppet/ssl]
Debug: /File[/var/lib/puppet/preview]: Autorequiring File[/var/lib/puppet]
Debug: /File[/var/lib/puppet/ssl/certs/ca.pem]: Autorequiring File[/var/lib/
puppet/ssl/certs]
Debug: Finishing transaction 69854562006000
Debug: Creating new connection for https://client.cn7788.com:8139
Error: Host client.cn7788.com failed: Error 403 on SERVER: Forbidden request:
server.cn7788.com(192.168.1.205) access to /run/client.cn7788.com [save]
authenticated at :119
```

最后, 在 puppet-client 端重启 Puppet 服务, 命令如下所示:

```
service puppet restart
```

(3) 在 server.cn7788.com 机器的 /etc 目录下建立 rsyncd.pass 文件并分配内容, 注意: 这个是推送到客户端的文件, 需要注意与 /etc/rsyncd.password 文件进行区分, /etc/rsyncd.pass 文件只需要指定同步用户的密码即可, /etc/rsyncd.password 文件内容如下所示:

```
test:test101
```

/etc/rsyncd.pass 文件内容如下所示:

```
test101
```

(4) 配置 Puppet 服务器端的 rsync 服务, /etc/rsyncd.conf 文件内容如下所示:

```
uid = www
gid = www
user chroot= no
max connections =200
timeout = 600
pid file = /var/run/rsyncd.pid
lock file = /var/run/rsyncd.lock
log file = /var/log/rsyncd.log

[www]
path=/var/www/html/
ignore errors
read only = no
list = no
hosts allow = 192.168.1.0/255.255.255.0
auth users = test
secrets file = /etc/rsyncd.password
```

我们的 Apache 服务的属主和属组是 www:www, 让 rsync 也以 www 用户运行, 这样可以保证通过 rsync 同步过去的文件属性。我这里采用 xinetd 管理 rsync, 将其中的 disable 改为 no, 并重启 xinetd 进程, 如下所示:

```
service xinetd restart
```

到了这一步其实还要仔细检查一下, 有时因为配置文件的错误或文件权限分配的错误, rsync 进程其实没有正确启动, 所以我们还要用如下命令来检查下:

```
lsof -i:873
```

命令显示结果如下, 这表明 rsync 进程已经在监听 873 端口了, 服务已经被正确启动起来了:

```
COMMAND  PID USER  FD   TYPE DEVICE SIZE/OFF NODE NAME
xinetd   7008 root   5u    IPv4  24249      0t0  TCP *:rsync (LISTEN)
```

(5) 创建名为 rsyncfile 的模块, 命令如下所示:

```
mkdir -p /etc/puppet/modules/wwwrsync/{manifests,files,templates}
```

并将 /etc/rsyncd.pass 复制到 /etc/puppet/modules/wwwrsync/files 目录下, 命令如下所示:

```
cp /etc/rsyncd.pass /etc/puppet/modules/wwwrsync/files/
```

(6) 我们在 `/etc/puppet/modules/wwwrsync/manifests/init.pp` 里定义一个名为 `wwwrsync` 的类, `init.pp` 文件内容如下所示:

```
class wwwrsync{
  package { httpd:
    ensure => present,
  }

  file {
    "/etc/rsyncd.pass":
    source =>"puppet://server.cn7788.com/modules/wwwrsync/rsyncd.pass",
    owner =>"root",
    group =>"root",
    mode =>"600",
  }

  exec {
    "auto rsync web directory":
    command =>"rsync -vzrtopg --delete test@192.168.1.205::www /var/www/html
    --password-file=/etc/rsyncd.pass",
    user =>"root",
    path => ["/usr/bin", "/usr/sbin", "/bin", "/bin/sh"],
  }
}
```

`init.pp` 文件包含了名为 `wwwrsync` 的类, 此类包含了三个资源, 第一个是名为 `httpd` 的资源包, 如果此服务没有安装的话, Puppet 客户端会自行安装 `httpd` 服务, 保证在本机自动生成 `/var/www/html` 目录; 第二个是 `file` 和 `exec` 命令, 它会将 `/etc/puppet/modules/wwwrsync/files/rsyncd.pass` 文件推送到 `puppet-client` 端后, 在 `puppet-client` 端执行 `rsync` 同步命令, 达到同步 `/var/www/html` 目录的目的, 所以 `rsync-vzrtopg` 后面应该接 `rsync` 服务器地址, 即 `192.168.1.205`, 这点请大家注意不要产生混淆。



**注意** `wwwrsync` 模块中定义的 `wwwrsync` 类要跟 `wwwrsync` 模块同名, 不然 Puppet 客户端在连接服务器端时会产生找不到 `wwwrsync` 类名的报错, 实验过程中如果遇到错误, 请注意多查看 Puppet 和系统日志。

(7) 我们接着在 `/etc/puppet/manifests/site.pp` 中定义一个 `default` 的特殊节点, 这是一个默认节点, 它会将 `wwwrsync` 类中的内容应用到所有主机上面, 其内容如下所示:

```
node default {
  include wwwrsync
}
```

(8) 我们在 `server.cn7788.com` 上面执行推送命令, 命令如下所示:

```
puppet kick -d --host `cat /etc/puppet/iplist.txt`
```

命令结果如下所示（反馈结果太长，这里只截取部分）：

```
Debug: /File[/var/lib/puppet/ssl/certs]/selrange: Found selrange default 's0'
for /var/lib/puppet/ssl/certs
Debug: /File[/var/lib/puppet/ssl/certs/ca.pem]: Autorequiring File[/var/lib/
puppet/ssl/certs]
Debug: /File[/var/lib/puppet/ssl/public_keys/server.cn7788.com.pem]:
Autorequiring File[/var/lib/puppet/ssl/public_keys]
Debug: /File[/var/lib/puppet/ssl/certs/server.cn7788.com.pem]: Autorequiring
File[/var/lib/puppet/ssl/certs]
Debug: /File[/var/lib/puppet/ssl/private_keys]: Autorequiring File[/var/lib/
puppet/ssl]
Debug: /File[/var/lib/puppet/lib]: Autorequiring File[/var/lib/puppet]
Debug: /File[/var/lib/puppet/ssl/private_keys/server.cn7788.com.pem]:
Autorequiring File[/var/lib/puppet/ssl/private_keys]
Debug: /File[/var/lib/puppet/ssl/private]: Autorequiring File[/var/lib/puppet/ssl]
Debug: /File[/var/lib/puppet/ssl/crl.pem]: Autorequiring File[/var/lib/puppet/ssl]
Debug: /File[/var/lib/puppet/ssl/certificate_requests]: Autorequiring File[/
var/lib/puppet/ssl]
Debug: /File[/var/lib/puppet/state]: Autorequiring File[/var/lib/puppet]
Debug: /File[/var/lib/puppet/ssl]: Autorequiring File[/var/lib/puppet]
Debug: /File[/var/lib/puppet/ssl/certs]: Autorequiring File[/var/lib/puppet/ssl]
Debug: /File[/var/lib/puppet/facts.d]: Autorequiring File[/var/lib/puppet]
Debug: /File[/var/lib/puppet/ssl/public_keys]: Autorequiring File[/var/lib/
puppet/ssl]
Debug: /File[/var/lib/puppet/preview]: Autorequiring File[/var/lib/puppet]
Debug: Finishing transaction 69995814564700
Debug: Creating new connection for https://client.cn7788.com:8139
Getting status
status is success
client.cn7788.com finished with exit code 0
```

/etc/puppet/iplist.txt 文件内容如下所示：

```
client.cn7788.com
nginx.cn7788.com
fabric.cn7788.com
```

我们观察名为 client.cn7788.com 的节点机器，它的 /var/www/html 文件立即就跟 server.cn7788.com 的 /var/www/html 目录同步了，从而实现了此需求，我们用 tail 命令观察 test1.cn7788.com 机器的 messages 日志，结果如下所示：

```
Nov 5 03:26:46 client puppet-agent[27782]: (/Stage[main]/Wwrsync/Exec[auto
rsync web directory]/returns) executed successfully
Nov 5 03:26:47 client puppet-agent[27782]: Finished catalog run in 0.74
seconds
Nov 5 03:41:50 client puppet-agent[28422]: (/Stage[main]/Wwrsync/Exec[auto
rsync web directory]/returns) executed successfully
```

```

Nov  5 03:41:51 client puppet-agent[28422]: Finished catalog run in 1.32
seconds
Nov  5 04:11:50 client puppet-agent[28690]: (/Stage[main]/Wwwrsync/Exec[auto
rsync web directory]/returns) executed successfully
Nov  5 04:11:50 client puppet-agent[28690]: (/Stage[main]/Wwwrsync/File[/etc/
rsyncd.pass]/content) content changed '{md5}d8e8fca2dc0f896fd7cb4cb0031ba249'
to '{md5}93412aea2e70977a362530b0dba2498a'
Nov  5 04:11:52 client puppet-agent[28690]: Finished catalog run in 1.97
seconds
Nov  5 04:14:17 client puppet-agent[27782]: triggered run
Nov  5 04:14:25 client puppet-agent[27782]: (/Stage[main]/Wwwrsync/Exec[auto
rsync web directory]/returns) executed successfully
Nov  5 04:14:26 client puppet-agent[27782]: Finished catalog run in 1.69
seconds

```

## 4.7 如何利用 ERB 模板自动配置 Apache 虚拟主机

线上环境有不少 Apache 主机需要增加基于域名的虚拟主机，特别是有的机器，虚拟主机达到几十台之多，这种情况下，如何能够方便快速地部署 httpd.conf 和虚拟主机配置文件呢？

这里就需要用到 Puppet 的 ERB 模板功能了，模板文件就是在模块下面 templates 目录中以“.erb”结尾的文件，Puppet 模板主要用于文件，例如各种服务的配置文件，相同的服务，不同的配置就可以考虑使用模板文件，例如 Nginx 和 Apache 的虚拟主机配置就可以考虑采用 ERB 模板的方案，我们在这里先以 Apache 为例来说明下，其模块的目录树结构如下：

```

/etc/puppet/
├── auth.conf
├── environments
│   └── example_env
│       ├── manifests
│       ├── modules
│       └── README.environment
├── fileserver.conf
├── manifests
│   ├── nodes
│   │   ├── client.pp
│   │   └── nginx.pp
│   └── site.pp
├── modules
│   └── apache
│       ├── files
│       ├── manifests
│       │   └── init.pp
│       └── templates
│           └── httpd.conf.erb
└── puppet.conf

```

11 directories, 9 files



Apache 模块的具体配置步骤如下。

(1) 首先我们建立一个 Apache 模块，命令如下所示：

```
mkdir -p /etc/puppet/modules/apache/{files,manifests,templates}
```

(2) /etc/puppet/modules/apache/manifests/init.pp 文件内容如下所示：

```
class apache{
    package{"httpd":
        ensure =>present,
    }

    service{"httpd":
        ensure      =>running,
        require     =>Package["httpd"],
    }
}

define apache::vhost ( $sitedomain, $rootdir,$port ) {
    file { [ "/etc/httpd/conf.d/httpd_vhost_${sitedomain}.conf":
        #path      => '/etc/httpd/conf/httpd_vhost.conf',
        content => template("apache/httpd.conf.erb"),
        require => Package["httpd"],
    ]
}
```

这里用到了 Puppet 中 `define` (定义) 的概念，定义和类都属于资源容器，不过定义的特点是能够在一台主机上被赋值多次，此外它还能接受参数。类在 Puppet 中是单例的，它们能够在一台主机节点机器上被包含多次，但是只会被求值一次；而定义因为能够接受参数，所以可以被声明多次，并且每一个声明都会被求值。

(3) /etc/puppet/modules/apache/templates 中的 `httpd.conf.erb` 模板文件内容如下所示：

```
<VirtualHost *:<%= port %>>
ServerName <%= sitedomain %>
DocumentRoot /var/www/html/<%= rootdir %>
    <Directory <%= rootdir %>>
        Options Indexes FollowSymLinks
        AllowOverride None
        Order allow,deny
        Allow from all
    </Directory>
ErrorLog logs/<%= sitedomain %>_error.log
CustomLog logs/<%= sitedomain %>_access.log common
</VirtualHost>
```

在 `httpd.conf.erb` 中我们提前定义了两个变量 `$sitedomain`、`$port`，Puppet 中使用这种格式 `<%= 变量名 %>` 来定义变量，我们可以检测一下模板是否存在语法问题，命令如下：

```
erb -x -T '-' -P /etc/puppet/modules/apache/templates/httpd.conf.erb | ruby -c
```

结果显示如下，表示语法不存在任何问题：

Syntax OK

(4) server 机器的 /etc/puppet/manifests/site.pp 内容如下：

```
import 'nodes/*.pp'
```

其下目录 nodes 中有两个文件：一个为 client.pp，另一个为 nginx.pp。server 机器的 /etc/puppet/manifests/nodes/nginx.pp 文件内容如下：

```
node 'nginx.cn7788.com' {
    include apache
    apache::vhost {'webmaster.cn7788.com':
        sitedomain => "webmaster.cn7788.com",
        rootdir => webmaster,
        port => 80,
    }
```

```
    apache::vhost {'webtest.cn7788.com':
        sitedomain => "webtest.cn7788.com",
        rootdir => webtest,
        port => 80,
    }
```

```
    apache::vhost {'webrsync.cn7788.com':
        sitedomain => "webrsync.cn7788.com",
        rootdir => webrsync,
        port => 80,
    }
}
```

另一台节点机器 client.cn7788.com 的 client.pp 配置文件内容，如下所示：

```
node 'client.cn7788.com' {
    include apache
    apache::vhost {'clientmaster.cn7788.com':
        sitedomain => "webmaster.cn7788.com",
        rootdir => webmaster,
        port => 80,
    }
```

```
    apache::vhost {'clientttest.cn7788.com':
        sitedomain => "webtest.cn7788.com",
        rootdir => webtest,
        port => 80,
    }
}
```

我们在 `nginx.cn7788.com` 的机器上输入如下命令进行验证:

```
puppet agent --test --server server.cn7788.com
```

结果如下所示:

```
Info: Retrieving pluginfacts
Info: Retrieving plugin
Info: Caching catalog for nginx.cn7788.com
Info: Applying configuration version '1446792027'
Notice: /Stage[main]/Apache/Package[httpd]/ensure: created
Notice: /Stage[main]/Main/Node[nginx.cn7788.com]/Apache::Vhost[webtest.cn7788.com]/File[/etc/httpd/conf.d/httpd_vhost_webtest.cn7788.com.conf]/ensure: defined content as '{md5}d5befc97a115a5069b6f8fd7e904b919'
Notice: /Stage[main]/Main/Node[nginx.cn7788.com]/Apache::Vhost[webmaster.cn7788.com]/File[/etc/httpd/conf.d/httpd_vhost_webmaster.cn7788.com.conf]/ensure: defined content as '{md5}07017828a5d085223c6635afffd0f69'
Notice: /Stage[main]/Apache/Service[httpd]/ensure: ensure changed 'stopped' to 'running'
Info: /Stage[main]/Apache/Service[httpd]: Unscheduling refresh on Service[httpd]
Notice: /Stage[main]/Main/Node[nginx.cn7788.com]/Apache::Vhost[webrsync.cn7788.com]/File[/etc/httpd/conf.d/httpd_vhost_webrsync.cn7788.com.conf]/ensure: defined content as '{md5}5fa6f92aa7ebb670a08e6e0968df9be6'
Notice: Finished catalog run in 87.65 seconds
```

结果表示配置是成功的, 整个过程耗时 87.65 秒。



**注意** 很多资料和文档都是复制 `/etc/httpd/conf/httpd.conf` 文件来作为 `httpd.conf.erb` 模板的, 我觉得这种做法还是欠缺考虑的, 一般来说, 每台 `Aapche` 主机上面至少有一个基于域名的虚拟主机, 有的更多, 十几个也很常见, 所以我们才需要用独立的虚拟主机文件来管理虚拟主机并自动载入 (注意配置文件 `httpd.conf` 中存在着这么一行 `include conf.d/*.conf`, 这个指令的意思是指将 `conf.d` 目录下所有以 `.conf` 结尾的文件都引进来), 这也是我们利用 ERB 模板文件将虚拟主机文件的定义路径放在 `/etc/httpd/conf.d` 目录下的原因。

## 4.8 如何利用 ERB 模板自动配置 Nginx 虚拟主机

对于已经上线的 Web 集群环境, 如何才能方便快速地部署 Nginx 及其虚拟主机呢? 要想实现这个需求我们可以参考 4.7 节的内容, 在这里建议用第三方 yum 源来安装 Nginx, 如果是用 Nginx 的官方源来安装 Nginx 的话, 我们可以查看下 `/etc/yum.repos.d/nginx.repo` 文件的内容, 如下所示:

```
[nginx]
name=nginx repo
baseurl=http://nginx.org/packages/centos/$releasever/$basearch/
gpgcheck=0
enabled=1
```

第二种方式是通过 `createrepo` 建立自己的 yum 源，这种方式更加灵活，我们可以在 Nginx 官网上下载适合自己的 rpm 源码包，然后 `rebuild` 成 rpm 包添加进自己的 yum 源，在自动化运维要求严格的定制环境中，绝大多数运维人员都会选择这种方法。大家通过这种方式安装 Nginx 后会发现，确实比用源码安装 Nginx 方便多了，就像自动分配了运行 Nginx 的用户 `nginx:nginx`。Nginx 的日志存放会被自动保存在 `/var/log/nginx` 下，其工作目录为 `/etc/nginx`，这一点跟源码编译安装的 Nginx 区别很大，请大家注意甄别。

Puppet 中的 `server.cn7788.com` 机器的 `/etc/puppet` 的文件结构如下所示：

```
|— auth.conf
|— environments
|   |— example_env
|       |— manifests
|       |— modules
|       |— README.environment
|— fileserver.conf
|— manifests
|   |— nodes
|       |— client.cn7788.com.pp
|       |— nginx.cn7788.com.pp
|   |— site.pp
|— modules
|   |— nginx
|       |— files
|       |— manifests
|       |   |— init.pp
|       |— templates
|       |   |— nginx.conf.erb
|       |   |— nginx_vhost.conf.erb
|— puppet.conf
```

(1) 首先我们要建立 Nginx 模块，命令如下所示：

```
mkdir -p /etc/puppet/modules/nginx/{files,manifests,templates}
```

(2) Nginx 模块的配置文件挺多的，这里将贴出其详细的配置说明。

`site.pp` 的文件内容如下：

```
import "nodes/*.pp"
```

`client.cn7788.com.pp` 的文件内容如下所示：

```
node 'client.cn7788.com' {
```

```

include nginx
nginx::vhost {'client.cn7788.com':
  sitedomain => "client.cn7788.com" ,
  rootdir => "client",
}
}

```

nginx.cn7788.com.pp 的文件内容如下所示:

```

node 'nginx.cn7788.com' {
  include nginx
  nginx::vhost {'nginx.cn7788.com':
    sitedomain => "nginx.cn7788.com" ,
    rootdir => "nginx",
  }
}

```

/etc/puppet/modules/nginx/manifests/init.pp 文件内容如下所示:

```

class nginx{
  package{"nginx":
    ensure =>present,
  }
  service{"nginx":
    ensure =>running,
    require =>Package["nginx"],
  }
  file{"nginx.conf":
    ensure => present,
    mode => 644,
    owner => root,
    group => root,
    path => "/etc/nginx/nginx.conf",
    content=> template("nginx/nginx.conf.erb"),
    require=> Package["nginx"],
  }
  define nginx::vhost($sitedomain,$rootdir) {
    file{ "/etc/nginx/conf.d/${sitedomain}.conf":
      content => template("nginx/nginx_vhost.conf.erb"),
      require => Package["nginx"],
    }
  }
}

```

/etc/puppet/modules/nginx/templates/nginx.conf.erb 的文件内容如下所示:

```

user nginx;
worker_processes 8;
error_log /var/log/nginx/error.log warn;
pid /var/run/nginx.pid;
events {

```

```

    use epoll;
    worker_connections 51200;
}
http {
    include      /etc/nginx/mime.types;
    default_type application/octet-stream;
    log_format   main '$remote_addr - $remote_user [$time_local] "$request" '
                    '$status $body_bytes_sent "$http_referer" '
                    '"$http_user_agent" "$http_x_forwarded_for"';
    access_log   /var/log/nginx/access.log main;
    sendfile     on;
    #tcp_nopush  on;
    keepalive_timeout 65;
    #gzip        on;
    include /etc/nginx/conf.d/*.conf;
}

```

我们检查下此 ERB 模板文件的语法，命令如下所示：

```
erb -x -T '-' -P /etc/puppet/modules/apache/templates/nginx.conf.erb | ruby -c
```

没有任何显示，这就说明文件在语法上是不存在任何问题的。

`/etc/puppet/modules/nginx/templates/nginx_vhost_conf.erb` 文件是 Nginx 虚拟主机的 ERB 模板，其文件内容如下所示：

```

server {
    listen      80;
    server_name <%= sitedomain %>;
    access_log  /var/log/nginx/<%= sitedomain %>.access.log;
    location / {
        root /var/www/<%= rootdir %>;
        index index.php index.html index.htm;
    }
}

```

最后我们可以在节点名为 `client.cn7788.com` 和 `nginx.cn7788.com` 的机器上验证效果，命令如下所示：

```
puppet agent --test --server server.cn7788.com
```

这里以 `nginx.cn7788.com` 进行举例说明，上述命令执行结果如下：

```

Info: Retrieving pluginfacts
Info: Retrieving plugin
Info: Caching catalog for nginx.cn7788.com
Info: Applying configuration version '1446798263'
Notice: /Stage[main]/Nginx/Package[nginx]/ensure: created
Notice: /Stage[main]/Nginx/Service[nginx]/ensure: ensure changed 'stopped' to
'running'
Info: /Stage[main]/Nginx/Service[nginx]: Unscheduling refresh on Service[nginx]

```

```
Notice: /Stage[main]/Main/Node[nginx.cn7788.com]/Nginx::Vhost[nginx.cn7788.com]/File[/etc/nginx/conf.d/nginx.cn7788.com.conf]/ensure: defined content as '{md5}5f08d10788e3c82b41336a40edc5350f'
```

```
Notice: /Stage[main]/Nginx/File[nginx.conf]/content:
```

```
--- /etc/nginx/nginx.conf2015-04-21 15:34:33.000000000 +0000
+++ /tmp/puppet-file20151106-5957-1f964a8-02015-11-06 08:27:14.267072983 +0000
@@ -1,32 +1,22 @@
```

```
-
-   user nginx;
-   worker_processes 1;
-
+   worker_processes 8;
+   error_log /var/log/nginx/error.log warn;
+   pid /var/run/nginx.pid;
-
-   events {
-       worker_connections 1024;
+       use epoll;
+       worker_connections 51200;
+   }
-
-   http {
+       include /etc/nginx/mime.types;
+       default_type application/octet-stream;
-
-       log_format main '$remote_addr - $remote_user [$time_local] "$request" '
+           '$status $body_bytes_sent "$http_referer" '
+           '"$http_user_agent" "$http_x_forwarded_for"';
-
-       access_log /var/log/nginx/access.log main;
-
-       sendfile on;
-       #tcp_nopush on;
-
-       keepalive_timeout 65;
-
-       #gzip on;
-
-       include /etc/nginx/conf.d/*.conf;
+   }
+ }
```

```
Info: Computing checksum on file /etc/nginx/nginx.conf
```

```
Info: FileBucket got a duplicate file {md5}f7984934bd6cab883e1f33d5129834bb
```

```
Info: /Stage[main]/Nginx/File[nginx.conf]: Filebucketed /etc/nginx/nginx.conf
to puppet with sum f7984934bd6cab883e1f33d5129834bb
```

```
Notice: /Stage[main]/Nginx/File[nginx.conf]/content: content changed '{md5}f798
4934bd6cab883e1f33d5129834bb' to '{md5}34e85800459aaf9b40ebfbdafa33614c0'
```



```
Notice: Finished catalog run in 42.19 seconds
```

我们在 [nginx.cn7788.com](http://nginx.cn7788.com) 的机器上检查生成的 Nginx 相关配置文件，发现都已经顺利生成了，这就说明 Nginx 模板配置是成功的。

## 4.9 小结

集中配置管理工具 Puppet 这个软件越来越成熟和强大了，它有着很好的发展前景，由于时间和环境的关系，本章只是简单介绍了自动化部署管理工具 Puppet 的安装、部署及平时工作中的常见用法，像 Puppet 的控制台产品 Dashboad 和 Foreman 都没有涉及，有兴趣的朋友可以结合工作实际尝试研究一下 Puppet 更高级的用法。

## 深度实践 iptables

### 作者简介

胥峰，盛大游戏高级研究员，《Linux 运维最佳实践》的作者。2006 年毕业于南京大学，2011 年加入盛大游戏。拥有十年运维经验，曾参与盛大游戏多款大型网络游戏的运维，主导统一运维平台的产品功能设计和实施，拥有工信部认证高级信息系统项目管理师资格。微信公众号“运维技术实践”、QQ 技术交流群群号 434242482。

在 Linux 运维工作中，每一个运维或许都使用过 iptables 进行网络安全设定。在涉及 Linux 安全时，我们或许会在第一时间想起 iptables 这个强大的工具。毋庸置疑的是，iptables 在 Linux 中具有重要的地位。但是，我们对这个工具是否还有更深入的理解呢？我们是否知道在使用 iptables 的过程中应该注意的事项呢？我们是否知道 iptables 除了在安全方面的作用之外，它还可以实现更多的功能呢？本章将会针对以上的问题，逐一进行详细阐述。

本章由一些使用 iptables 中的经典案例开始，对 iptables 的状态追踪功能进行生动细致的讲解，对在 iptables 中限制 ICMP 协议时的注意事项运用案例进行讲解，然后介绍 iptables 在非安全方面的功能：网络地址转换。最后，作为总结，我们将以一张图来展示 iptables 中的各种表和链的作用时间。

## 5.1 禁用连接追踪

### 5.1.1 排查连接追踪导致的故障

#### 1. 问题描述

虚拟机用户在测试网络连通性时，发现连接到某主机的网络 ping 时断时续，丢包问题严

重。同时，它在与该虚拟机同网段的 Windows 物理机上测试同一 IP 时，未发现该问题。

用户给我们提供的截图如图 5-1 所示。

图 5-1 ping 丢包截图

用户使用的测试脚本 testping.sh 内容如下：

```
#!/bin/bash

host=$1
wait=$2

if [ -z $host ]; then
    echo "Usage: `basename $0` [HOST]"
    exit 1
fi

if [ -z $wait ]; then
    wait=1
fi

let index=1
let lost=0
while ;; do
    result=`ping -W 1 -c 1 $host | grep 'bytes from '`
    if [ $? -gt 0 ]; then
        echo -e "$lost/$index - `date +%Y/%m/%d %H:%M:%S` - host $host is \033[0;31mdown\033[0m"
        let lost=$lost+1
    else
        echo -e "$lost/$index - `date +%Y/%m/%d %H:%M:%S` - host $host is \033[0;32mok\033[0m - `echo $result | cut -d ':' -f 2`"
        sleep $wait # avoid ping rain
    fi
    let index=$index+1
done
```

用户执行以下命令进行测试：

```
sh testping.sh xxx.yyy.zzz.76
```

## 2. 排查过程

在收到故障申报后，我们首先分析了系统日志 /var/log/messages，发现在对应的时间点，

有关于 `nf_conntrack` 的报错。报错内容如下：

```
Dec 30 17:19:02 gcloud-whcq-ISpeaker-198 kernel: __ratelimit: 877 callbacks
suppressed
Dec 30 17:19:02 gcloud-whcq-ISpeaker-198 kernel: nf_conntrack: table full,
dropping packet.
Dec 30 17:19:02 gcloud-whcq-ISpeaker-198 kernel: nf_conntrack: table full,
dropping packet.
Dec 30 17:19:02 gcloud-whcq-ISpeaker-198 kernel: nf_conntrack: table full,
dropping packet.
Dec 30 17:19:02 gcloud-whcq-ISpeaker-198 kernel: nf_conntrack: table full,
dropping packet.
Dec 30 17:19:02 gcloud-whcq-ISpeaker-198 kernel: nf_conntrack: table full,
dropping packet.
Dec 30 17:19:02 gcloud-whcq-ISpeaker-198 kernel: nf_conntrack: table full,
dropping packet.
Dec 30 17:19:02 gcloud-whcq-ISpeaker-198 kernel: nf_conntrack: table full,
dropping packet.
Dec 30 17:19:07 gcloud-whcq-ISpeaker-198 kernel: __ratelimit: 1356 callbacks
suppressed
Dec 30 17:19:07 gcloud-whcq-ISpeaker-198 kernel: nf_conntrack: table full,
dropping packet.
```

我们使用如下 2 个命令可以看出，服务器上的已有连接追踪条目数量已接近于我们配置的最大追踪数，这一点进一步确认了丢包问题是由连接追踪导致的。

```
# sysctl net.netfilter.nf_conntrack_max
net.netfilter.nf_conntrack_max = 65536
# sysctl net.netfilter.nf_conntrack_count
net.netfilter.nf_conntrack_count = 65000
```

### 5.1.2 分析连接追踪的原理

概要地说，连接追踪系统在一个内存数据结构中记录了连接的状态，这些信息包括源 IP、目的 IP、双方端口号（对 TCP 和 UDP）、协议类型、状态和超时信息等。有了这些信息，我们就可以设置更灵活的过滤策略。



**注意** 连接追踪系统本身不进行任何过滤动作，它只是为上层应用（如 iptables）提供基于状态的过滤功能。

下面来看一个实际的例子（通过 `cat/proc/net/nf_conntrack` 命令可以查看当前连接追踪的表）：

```
ipv4      2 tcp      6 62 SYN_SENT src=xxx.yyy.19.201 dst=87.240.131.117 sport=24943
dport=443 [UNREPLIED] src=87.240.131.117 dst=xxx.yyy.19.201 sport=443 dport=24943
mark=0 secmark=0 use=2
# 该条目的意思是：系统收到了来自 xxx.yyy.19.201:24943 发送到 87.240.131.117:443 的第一个
TCP SYN 包，但此时对方还没有回复这个 SYN 包（UNREPLIED）
ipv4      2 tcp      6 30 SYN_RECV src=106.38.214.126 dst=xxx.yyy.19.202 sport=18102
dport=6400 src=xxx.yyy.19.202 dst=106.38.214.126 sport=6400 dport=18102 mark=0
```

```
secmark=0 use=2
```

# 该条目的意思是：系统收到了来自 106.38.214.126:18102 发送到 xxx.yyy.19.202:6400 的第一个 TCP SYN 包

```
ipv4      2 tcp      6 158007 ESTABLISHED src=xxx.yyy.19.201 dst=211.151.144.188 sport=
48153 dport=80 src=211.151.144.188 dst=xxx.yyy.19.201 sport=80 dport=48153 [ASSURED]
mark=0 secmark=0 use=2
```

# 该条目的意思是：xxx.yyy.19.201:48153<-->211.151.144.188:80 之间的 TCP 连接是 ESTABLISHED 状态，这个连接是被保证的 (ASSURED，不会因为内存耗尽而丢弃)

该表中的数据提供的状态信息，可以使用 iptables 的 state 模块进行状态匹配，进而执行一定的过滤规则。目前 iptables 支持基于以下 4 种状态的过滤规则：INVALID、ESTABLISHED、NEW 和 RELATED。

启用连接追踪后，在某些情况下，设置 iptables 会变得比较简单，如图 5-2 所示。

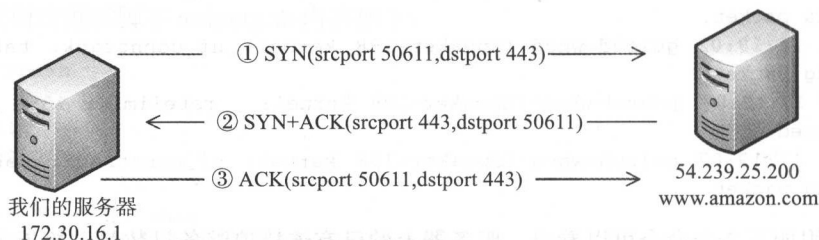


图 5-2 主动访问外网服务的 TCP3 次握手示意图

当我们的服务器需要主动访问 https://www.amazon.com 提供的接口时，3 次握手的过程如图 5-2 所示。在基于状态进行 iptables 设置时，使用如下的规则即可：

```
iptables -A INPUT -p tcp -m state --state ESTABLISHED -j ACCEPT # rule1
iptables -A OUTPUT -p tcp -j ACCEPT # rule2
```

工作流程如下：

(1) 第 1 个包①匹配到规则 rule2，允许。

(2) 第 2 个包②因为在 nf\_conntrack 表中有如下的规则匹配到 rule1，允许：

```
ipv4      2 tcp      6 431995 ESTABLISHED src=172.30.16.1 dst=54.239.25.200
sport=50611 dport=443 src=54.239.25.200 dst=172.30.16.1 sport=443 dport=50611
[ASSURED] mark=0 secmark=0 use=2
```

(3) 第 3 个包③匹配到规则 rule2，允许。

### 5.1.3 禁用连接追踪的方法

通过以上的分析，我们知道在进行大量网络传输连接的时候，启用连接追踪可能会导致网络丢包、TCP 重传等问题，因此，我们需要在适用的情况下禁用连接追踪。

禁用连接追踪的方法有 3 种，具体如下。

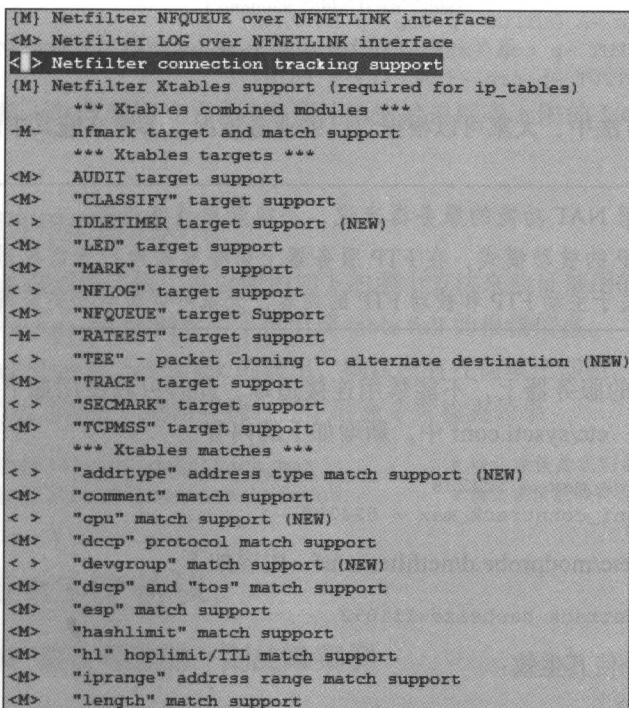
(1) 在内核中禁用 Netfilter Connection tracking support。

编译内核时，依次进入 Networking support → Networking options → Network packet filtering

framework (Netfilter) → Core Netfilter Configuration, 禁用的方法如图 5-3 所示 (取消选中 Netfilter connection tracking support)。

这样编译出来的内核, 将不支持连接追踪的功能, 也就是不会生成以下的 ko 文件了:

```
kernel/net/netfilter/nf_conntrack.ko
kernel/net/netfilter/nf_conntrack_proto_dccp.ko
kernel/net/netfilter/nf_conntrack_proto_gre.ko
kernel/net/netfilter/nf_conntrack_proto_sctp.ko
kernel/net/netfilter/nf_conntrack_proto_udplite.ko
kernel/net/netfilter/nf_conntrack_netlink.ko
kernel/net/netfilter/nf_conntrack_amanda.ko
kernel/net/netfilter/nf_conntrack_ftp.ko
kernel/net/netfilter/nf_conntrack_h323.ko
kernel/net/netfilter/nf_conntrack_irc.ko
kernel/net/netfilter/nf_conntrack_broadcast.ko
kernel/net/netfilter/nf_conntrack_netbios_ns.ko
kernel/net/netfilter/nf_conntrack_snmp.ko
kernel/net/netfilter/nf_conntrack_pptp.ko
kernel/net/netfilter/nf_conntrack_sane.ko
kernel/net/netfilter/nf_conntrack_sip.ko
kernel/net/netfilter/nf_conntrack_tftp.ko
kernel/net/netfilter/xt_conntrack.ko
kernel/net/ipv4/netfilter/nf_conntrack_ipv4.ko
kernel/net/ipv6/netfilter/nf_conntrack_ipv6.ko
```



```
[M] Netfilter NFQUEUE over NFNETLINK interface
<M> Netfilter LOG over NFNETLINK interface
< > Netfilter connection tracking support
[M] Netfilter Xtables support (required for ip_tables)
    *** Xtables combined modules ***
-M- nftmark target and match support
    *** Xtables targets ***
<M> AUDIT target support
<M> "CLASSIFY" target support
< > IDLETIMER target support (NEW)
<M> "LED" target support
<M> "MARK" target support
< > "NFLOG" target support
<M> "NFQUEUE" target Support
-M- "RATEEST" target support
< > "TEE" - packet cloning to alternate destination (NEW)
<M> "TRACE" target support
< > "SECMARK" target support
<M> "TCPMSS" target support
    *** Xtables matches ***
< > "addrtype" address type match support (NEW)
<M> "comment" match support
< > "cpu" match support (NEW)
<M> "dccp" protocol match support
< > "devgroup" match support (NEW)
<M> "dscp" and "tos" match support
<M> "esp" match support
<M> "hashlimit" match support
<M> "hl" hoplimit/TTL match support
<M> "iprange" address range match support
<M> "length" match support
```

图 5-3 编译内核时禁用连接追踪的方法

此时，在 iptables 中不能再使用 NAT 功能，同时也不能再使用 -m state 模块了，否则会产生如下的报错信息：

```
[root@localhost ~]# iptables -t nat -A POSTROUTING -o eth0 -s 172.30.4.0/24 -j SNAT --to 172.30.4.11
iptables v1.4.7: can't initialize iptables table `nat': Table does not exist (do you need to insmod?)
Perhaps iptables or your kernel needs to be upgraded.
[root@localhost ~]# iptables -I INPUT -p tcp -m state --state NEW -j ACCEPT
iptables: No chain/target/match by that name.
```

(2) 在 iptables 中，禁用 -m state 模块，同时在 filter 表的 INPUT 链中显式指定 ACCEPT。以图 5-2 为例，在满足这样的访问需求时，我们使用的 iptables 必须修改为如下的内容：

```
iptables -A INPUT -p tcp -s 54.239.25.200 --sport 443 -j ACCEPT # rule1
iptables -A OUTPUT -p tcp -j ACCEPT # rule2
```

同时，在 /etc/init.d/iptables 中，修改如下的内容：

```
修改前：NF_MODULES_COMMON=(x_tables nf_nat nf_conntrack) # Used by netfilter v4 and v6
修改后：NF_MODULES_COMMON=(x_tables) # Used by netfilter v4 and v6
```

(3) 在 iptables 中，使用 raw 表指定 NOTRACK。

```
iptables -t raw -A PREROUTING -p tcp -j NOTRACK
iptables -t raw -A OUTPUT -p tcp -j NOTRACK
iptables -A INPUT -p tcp -s 54.239.25.200 --sport 443 -j ACCEPT # rule1
iptables -A OUTPUT -p tcp -j ACCEPT # rule2
```

在以上的 3 种方法中，大家可以根据自己的业务情况，参考实施其中的一种。



**注意** (1) 对于使用 NAT 功能的服务器来说，不能禁用连接追踪。

(2) 对于 FTP 的被动模式，在 FTP 服务器上需要显式地打开需要进行数据传输的端口范围。关于主动 FTP 和被动 FTP 的内容，大家可以参考相关资料，本书不再赘述。

在配置了 NAT 的服务器上，不能禁用连接追踪，此时可以使用如下的方法来提高连接追踪的条目上限。在 /etc/sysctl.conf 中，新增如下的内容：

```
net.nf_conntrack_max = 524288
net.netfilter.nf_conntrack_max = 524288
```

新增配置文件 /etc/modprobe.d/netfilter.conf，内容如下：

```
options nf_conntrack hashsize=131072
```

执行以下的命令使其生效：

```
/etc/init.d/iptables restart # 重新加载连接追踪模块，同时更新 nf_conntrack 配置 hashsize
sysctl -p # 使得修改的 sysctl.conf 中 nf_conntrack 的上限提高
```



系统 `nf_conntrack_max` 的值，在未指定时，可根据以下公式计算得出：

$$\text{nf\_conntrack\_max} = \text{nf\_conntrack\_buckets} * 4$$

系统 `nf_conntrack_buckets` 的值，在未指定时，可根据以下公式计算得出：

在系统内存大于等于 4GB 时，`nf_conntrack_buckets` = 65536

在系统内存小于 4GB 时，`nf_conntrack_buckets` = 内存大小 / 16384

在本案例中，我们使用 `options nf_conntrack hashsize=131072` 自主指定了 Buckets 的大小。

Buckets 和连接追踪表的关系如图 5-4 所示。

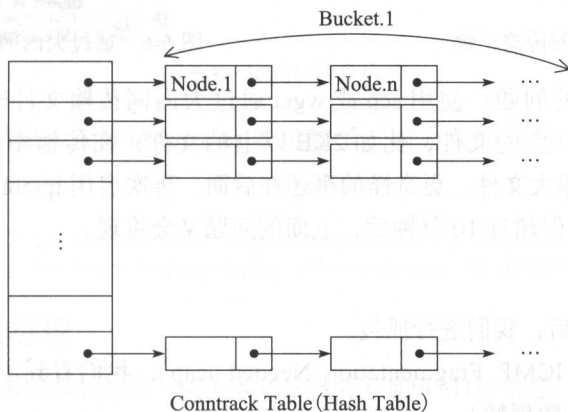


图 5-4 Buckets 和连接追踪表的关系

为 Buckets 设置一个合理的值（一般为预计的连接追踪表上限的 1/4），使得连接追踪表的定位效率最高。

#### 5.1.4 确认禁用连接追踪的效果

我们在禁用了连接追踪之后，可以使用如下的两个方法来验证禁用的效果：

(1) 确认 `/var/log/messages` 内容中不再出现 `table full` 的报错信息。

(2) 检查 `lsmod | grep nf_conntrack` 的输出，确认没有任何输出即可。

如果是在 NAT 服务器上，则需要执行以下的命令来检查效果：

```
sysctl net.netfilter.nf_conntrack_max      # 确认该值是我们修改后的结果
sysctl net.netfilter.nf_conntrack_count    # 确认该值能够突破出现问题的最大追踪数
```

## 5.2 慎重禁用 ICMP 协议

### 5.2.1 禁用 ICMP 协议导致的一则故障案例

#### 1. 问题描述

我们负责维护的某系统分布于多个机房之中，之前文件传输一直走的公网，这是很正常

的（架构如图 5-5 所示）。

后来建成了大内网（使用 GRE VPN 技术在 Internet 上组建的私有网络，目前通过北京电信通转发，使电信和联通之间的互访变得更快速），为了安全和快速，文件传输改走大内网（架构如图 5-6 所示）。

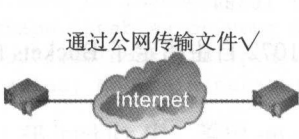


图 5-5 通过公网传输正常



图 5-6 通过大内网传输失败

结果碰到了奇怪的问题：使用 scp 或 wget 通过大内网传输文件时，只能传输 1KB 大小左右的文件，稍大一点的文件，比如 2KB 以上的文件，在传输中就被卡住了。当停止 iptables 后，则可以传输大文件。更奇怪的事还在后面，再次启用 iptables，大约在 10 分钟内仍然可以传输大文件，但超过 10 分钟后，上面的问题又会重现。

2. 排查过程

在启动 iptables 之后，我们进行抓包。

在抓包中（文件：ICMP\_Fragmentation\_Needed.pcap），我们看到了有 ICMP 的报错信息（如图 5-7 中的❶所示的数据帧）。

No.	Time	Source	Destination	Protocol	Length	Info
15	5.867249	10.2.3.27	10.10.60.69	TCP	66	42711 > 80 [SYN] Seq=1654818306 win=5840 Len=0 MSS=
16	5.867264	10.10.60.69	10.2.3.27	TCP	66	80 > 42711 [SYN, ACK] Seq=563863027 Ack=1654818307 W
17	5.887236	10.2.3.27	10.10.60.69	TCP	60	42711 > 80 [ACK] Seq=1654818307 Ack=563863028 Win=6
18	5.887245	10.2.3.27	10.10.60.69	HTTP	186	GET /latale/13/data HTTP/1.0
19	5.887250	10.10.60.69	10.2.3.27	TCP	54	80 > 42711 [ACK] Seq=563863028 Ack=1654818439 Win=6
20	5.887350	10.10.60.69	10.2.3.27	TCP	2974	[TCP segment of a reassembled PDU]
21	5.887468	10.10.251.2	10.10.60.69	ICMP	70	Destination unreachable (Fragmentation needed)❶
27	8.887239	10.10.60.69	10.2.3.27	TCP	1514	[TCP Retransmission] 80 > 42711 [ACK] Seq=563863028
28	8.887343	10.10.251.2	10.10.60.69	ICMP	70	Destination unreachable (Fragmentation needed)
41	14.887861	10.10.60.69	10.2.3.27	TCP	1514	[TCP Retransmission] 80 > 42711 [ACK] Seq=563863028
42	14.887974	10.10.251.2	10.10.60.69	ICMP	70	Destination unreachable (Fragmentation needed)

图 5-7 ICMP 报错信息

编号为 20 的数据帧，是服务器 10.10.60.69 向网卡提交了长度为 2974 的数据帧（该网卡支持 TSO，进行自动分片传输到网络上）后，在编号为 21 的数据帧中，被路由器 10.10.251.2 返回了 ICMP Destination unreachable（Fragmentation needed）的信息。

ICMP 信息的具体内容如图 5-8 所示。

图 5-8 中，❶是 ICMP 类型，❷是该类型 ICMP 的错误代码，❸是通知 10.10.60.69（发送方）应该使用的 MTU（1400 字节），❹和❺是引起这个 ICMP 的数据帧的源 IP 和目的 IP（IP 层信息），❻和❼是引起这个 ICMP 的数据帧的源端口和目的端口（TCP 层信息），❽是引起这个 ICMP 的数据帧的 TCP 序列号。❸、❹、❺、❻、❼正好与图 5-7 中的编号为 20 的数据帧相匹配。

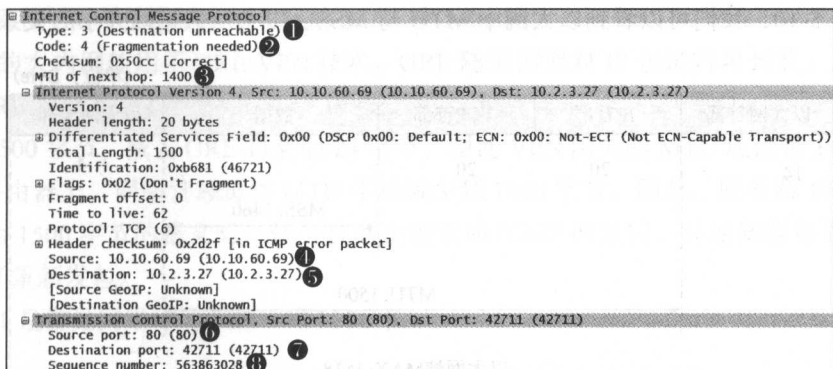


图 5-8 ICMP 信息内容

而这个 ICMP 的信息，正好被 iptables 给过滤了。因为我们只开放了 ICMP 的以下类型为允许：

```
iptables -A INPUT -p icmp --icmp-type echo-reply -j ACCEPT
iptables -A INPUT -p icmp --icmp-type echo-request -j ACCEPT
```

### 5.2.2 MTU 发现的原理

通过以上的分析，我们认识到有一类 ICMP 专门用于通知 MTU 信息，那么什么是 MTU 呢？

MTU (Maximum Transmission Unit, 最大传输单元) 是指一种通信协议的某一层所能通过的最大数据包大小 (以字节为单位)。由于以太网传输电气方面的限制，每个以太网帧最大不能超过 1518 字节，刨去以太网帧的帧头 (源 MAC+ 目标 MAC+Type+CRC) 18 字节，那么剩下承载上层协议的 Data 域 (IP 头+TCP 头+应用数据) 最大就只能是 1500 字节 (如图 5-9 所示)，我们称该值为 MTU。

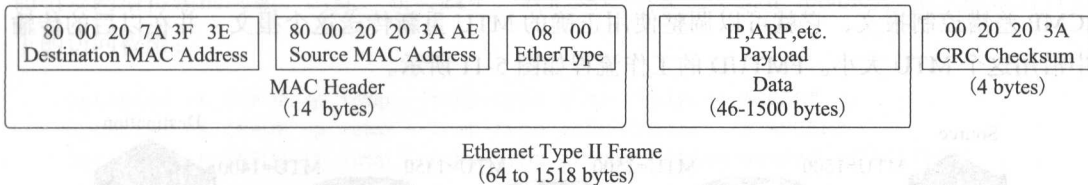


图 5-9 以太网数据帧封装结构图

以下是一些与 MTU 密切相关的概念。

GRE：通用路由封装协议 (Generic Routing Encapsulation)，规定了如何用一种网络协议去封装另一种网络协议的方法，它是一种应用非常广泛的第三层 VPN 隧道协议。

MSS：最大分段尺寸 (Maximum Segment Size)，是 TCP 数据包每次能够传输的最大数据分段。这个值等于 MTU 减去 IP 数据包包头的 20 字节和 TCP 数据段包头的 20 字节，所以 MSS 一般为 1460 字节。TCP 协议在建立连接的时候双方将会协商本次通信使用的 MSS 值。

通过图 5-10，我们可以看到以太网中 MTU 与 MSS、以太网数据帧大小的关系。

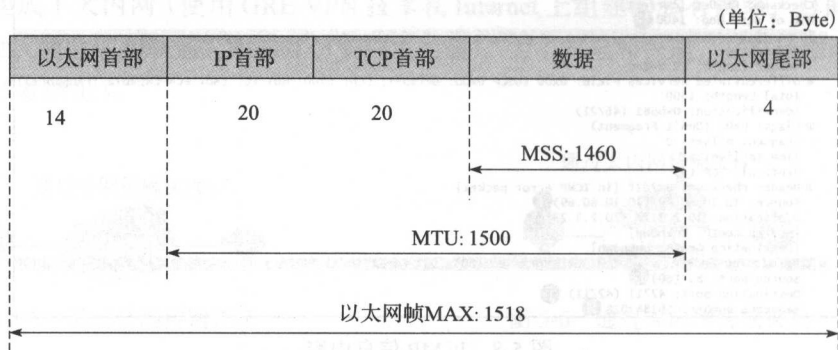


图 5-10 MTU 与 MSS、以太网数据帧关系图

**PMTU：**路径最大传输单元 (Path Maximum Transmission Unit)。在因特网上，两台主机之间的通信要经过多个网络，而每个不同的网络在 IP 层可能会有不同的 MTU。两台通信主机路径中的最小 MTU 被称作路径 MTU(PMTU)。默认情况下,PMTU 的老化时间是 10 分钟。我们可以通过配置 PMTU 的老化时间来更改 PMTU 项在缓存中的时间。在 Linux 中，我们可以使用如下的命令检查该老化时间：

```
sysctl net.ipv4.route.mtu_expires
```

**PMTUD：**路径最大传输单元发现 (Path MTU Discovery)。通过在 IP 首部中设置“不分片位 (Don't Fragment, DF)”，来确定当前路径的路由器是否需要正在发送的 IP 数据报进行分片。如果一个待转发的 IP 数据报被设置成 DF 比特，而其长度又超过了 MTU，那么路由器就会丢弃这个报文，并返回一个 ICMP 不可达的差错报文 (类型为 3、代码为 4：需要进行分片但设置了不分片位)，其中填有下一跳正确的 MTU。如果发送端主机接收到这个 ICMP 差错控制报文，它就可以调整使用正确的 MTU 重新传送这个报文，并在以后的传输中沿用这个 MTU 大小。PMTUD 的工作流程如图 5-11 所示。

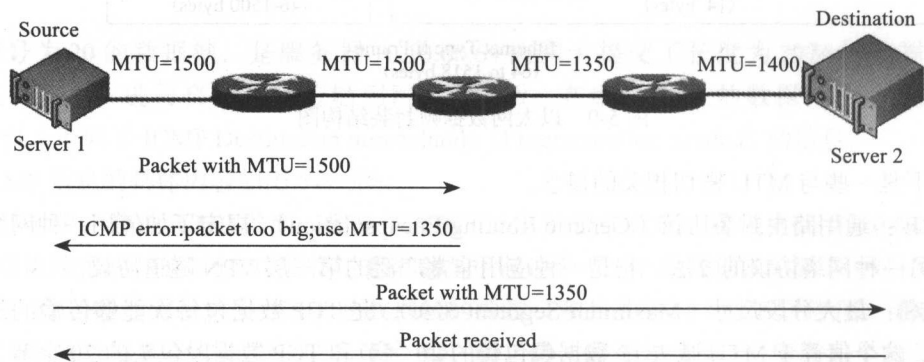


图 5-11 PMTUD 工作流程

为什么图 5-8 中的❷提示下一跳的 MTU 是 1400 呢？

公司的大内网使用了 GRE VPN 技术。GRE 隧道需要对 IP 包进行再封装，这样就额外增加了 GRE 报文头（4 字节）+ 外层 IP 报文头（20 字节），总共 24 字节。而以太网默认的 MTU 为 1500 字节，减去 GRE 封装的 24 字节，因此 VPN 网关的 MTU 应该是 1476 字节了。在 VPN 路由器上，网络管理员将 MTU 手动减少到 1400 字节。因此，服务器 10.10.60.69 发出 MTU 为 1500 字节的报文时，就会被路由器返回 ICMP 的报错，并通知服务器以 1400 字节的 MTU 重新发包。

表 5-1 是常见网络环境下的一些 MTU 值。

表 5-1 常见网络环境下的 MTU 值

MTU 值 (单位: Byte)	描述
1500	以太网信息包最大值，也是默认值
1492	PPPoE 的最佳值
1476	GRE VPN 的最大值
1472	使用 ping 的最大值
1468	DHCP 的最佳值
1430	VPN 和 PPTP 的最佳值
576	拨号连接到 ISP 的标准值

## 5.2.3 解决问题的方法

在 iptables 中，增加以下的条目：

```
iptables -A INPUT -p icmp --icmp-type fragmentation-needed -j ACCEPT
```

在这个案例中，我们可以看到，如果简单地禁止 iptables 会导致 MTU 协商不成功，从而引发网络传输的问题。因此，在实践中，我们建议不要完全禁止 iptables，至少应该打开以下的访问权限：

```
iptables -A INPUT -p icmp --icmp-type echo-reply -j ACCEPT
iptables -A INPUT -p icmp --icmp-type echo-request -j ACCEPT
iptables -A INPUT -p icmp --icmp-type fragmentation-needed -j ACCEPT
```

## 5.3 网络地址转换在实践中的案例

在实践中，iptables 除了可用于网络安全之外，还经常用于网络地址转换（NAT）的环境中。网络地址转换分为源地址转换（源地址 NAT）和目的地址转换（目的地址 NAT）。

### 5.3.1 源地址 NAT

源地址 NAT，主要用于如图 5-12 所示的网络示意图中无外网 IP 的服务器（Server B）需

要访问互联网的场景。

在图 5-12 中, Server B 没有外网 IP, 若其需要访问互联网, 则需要进行如下的设置。

(1) 在服务器 Server B 上, 指定其网络的默认网关是 10.128.70.112 (即 Server A 的内网地址)。

(2) 在服务器 Server A 上, 启用路由功能。启用的方法是执行以下的命令:

```
sysctl -w net.ipv4.ip_forward=1
```

(3) 在 Server A 上, 设置 iptables 规则如下:

```
iptables -t filter -A FORWARD -j ACCEPT
iptables -t nat -A POSTROUTING -o eth0 -j
SNAT --to x.y.z.173 #eth0 是 Server A 的外网网卡,
x.y.z.173 是 Server A 的外网 IP
```

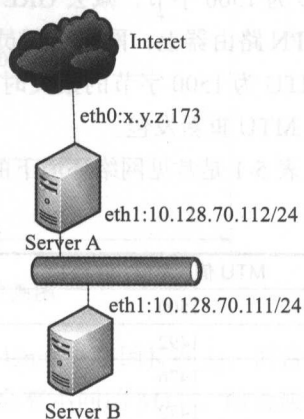


图 5-12 网络地址转换的网络示意图

经过以上 3 个步骤的设置之后, Server B 将会通过 Server A 访问互联网。此时, 在互联网上看到的源地址将是 Server A 的外网 IP。

以 Server B 访问 8.8.8.8 的 DNS 服务为例, 其数据流程如下。

(1) 在 Server B 上, 网络层数据包的格式为: 目的地址 IP 8.8.8.8, 源地址 IP 10.128.70.111。

(2) 在 Server A 上经过源地址 NAT 后的网络层数据包格式为: 目的地址 IP 8.8.8.8, 源地址 IP x.y.z.173。该转换条目被记录在 /proc/net/nf\_conntrack 中。

(3) 8.8.8.8 的响应 (源地址 IP 8.8.8.8, 目的地址 IP x.y.z.173) 到达 Server A 之后, Server A 将改写网络层数据包为源地址 IP 8.8.8.8, 目的地址 IP 10.128.70.111。

这就是源地址 NAT 的工作过程。

### 5.3.2 目的地址 NAT

目的地址 NAT 用于如图 5-12 所示的网络示意图中, 外部用户直接访问无外网 IP 的服务器 (Server B) 提供的服务。例如, 外部用户希望通过互联网访问到 Server B 上的 Oracle 数据库 (监听端口是 TCP 1521) 时, 我们可以使用如下的命令在 Server A 上进行目的地址 NAT 设置:

```
iptables -t nat -A PREROUTING -d x.y.z.173 -p tcp -m tcp --dport 1521 -j DNAT
--to-destination 10.128.70.111:1521 # 改写目的地址为 10.128.70.111, 目的端口为 1521
iptables -t nat -A POSTROUTING -d 10.128.70.111 -p tcp -m tcp --dport 1521 -j
SNAT --to-source 10.128.70.112 # 改写源地址 IP 为 Server A 的内网 IP, 此时与 Server B 相
当于是和 Server A 进行通信
```

网络地址转换是运维人员在工作中经常会用到的技术，我们需要非常熟悉源地址转换和目的地址转换这两种方案。

## 5.4 深入理解 iptables 的各种表和链

通过以上几节的实践我们知道，iptables 为系统工程师提供了强大的包过滤功能和 NAT 网络地址转换功能。在 Linux 中，为 iptables 提供这些功能的底层模块是 netfilter 框架。netfilter 是 Linux 内核中的一系列钩子（hook），它为内核模块在网络栈中的不同位置注册回调函数（callback function）提供了支持。数据包在协议栈中将依次经过这些处于不同位置的回调函数的处理。

下面我们以图 5-13 讲解 netfilter 钩子与 iptables 各种表和链的处理顺序：

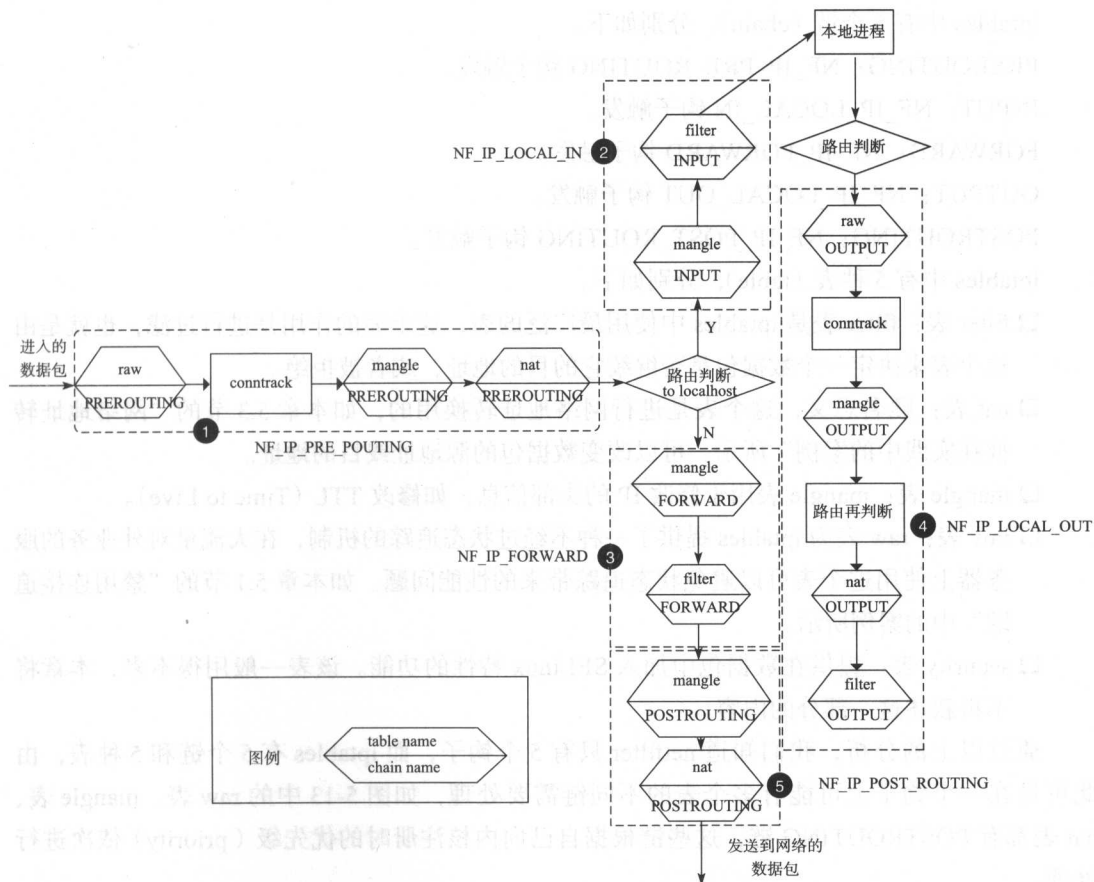


图 5-13 netfilter 钩子与 iptables 各种表和链的处理顺序图



netfilter 提供了 5 个钩子可以供程序去注册。在数据包经过网络栈的时候，这些钩子上注册的内核模块将依次被触发。这 5 个钩子的处理时间分别如下。

**NF\_IP\_PRE\_ROUTING**：在数据流量进入网络栈后立即被触发，这个钩子上注册的模块在路由决策前即被执行。如图 5-13 中❶所示的阶段。

**NF\_IP\_LOCAL\_IN**：在路由判断确定了包是发送到本机时执行这个钩子。如图 5-13 中❷所示的阶段。

**NF\_IP\_FORWARD**：在路由判断确定了包需要转发给其他主机时执行这个钩子。如图 5-13 中的❸所示的阶段。

**NF\_IP\_LOCAL\_OUT**：在本机进程产生的网络被送到网络栈上时执行这个钩子，如图 5-13 中❹所示的阶段。

**NF\_IP\_POST\_ROUTING**：在数据包经过路由判断后即将发送到网络前执行这个钩子。如图 5-13 中❺所示的阶段。

iptables 中有 5 个链 (chain)，分别如下。

**PREROUTING**：NF\_IP\_PRE\_ROUTING 钩子触发。

**INPUT**：NF\_IP\_LOCAL\_IN 钩子触发。

**FORWARD**：NF\_IP\_FORWARD 钩子触发。

**OUTPUT**：NF\_IP\_LOCAL\_OUT 钩子触发。

**POSTROUTING**：NF\_IP\_POST\_ROUTING 钩子触发。

iptables 中有 5 种表 (table)，分别如下。

❑ **filter 表**：filter 表是 iptables 中使用最广泛的表，这个表的作用是进行过滤，也就是由这个表来决定一个数据包是否继续它的目的地址，或者被拒绝。

❑ **nat 表**：顾名思义，这个表是进行网络地址转换用的，如本章 5.3 节的“网络地址转换在实践中的案例”所示，可以改变数据包的源地址或目的地址。

❑ **mangle 表**：mangle 表用于修改 IP 的头部信息，如修改 TTL (Time to Live)。

❑ **raw 表**：raw 表为 iptables 提供了一种不经过状态追踪的机制，在大流量对外业务的服务器上使用这个表可以避免状态追踪带来的性能问题。如本章 5.1 节的“禁用连接追踪”中的案例所示。

❑ **security 表**：提供在数据包中加入 SELinux 特性的功能。该表一般用得不多，本章将不再叙述这一部分的内容。

通过以上的分析，我们知道 netfilter 只有 5 个钩子，而 iptables 有 5 个链和 5 种表，由此可见在一个钩子上可能有多个表的不同链需要处理，如图 5-13 中的 raw 表、mangle 表、nat 表都有 POSTROUTING 链，这些链根据自己向内核注册时的优先级 (priority) 依次进行处理。

## 5.5 小结

Linux 中的 iptables 既是强大的网络安全工具，又是网络地址转换工具。本章重点剖析了连接追踪的机制及使用中的注意事项、提出了慎重禁用 ICMP 协议的论点并进行了分析。对 iptables 在网络地址转换中的两种经典使用场景进行了配置案例的说明。作为总结部分，我们给出了 iptables 中各种表和链的作用和关系，希望能帮读者形成一个清晰的图像，在分析 iptables 问题时能够参照这个图像进行快速定位。

## 使用 systemd 管理 Linux 系统服务

### 作者简介

尹会生，金山西山居高级系统工程师。拥有多年的企业集群解决方案培训经验和内核调优经验，擅长高性能和高可用性集群技术。近四年专注于 Hadoop 集群、Spark 集群在推荐系统和 BI 相关领域的解决方案。曾历任北京尚观科技高级讲师、新浪研发中心技术经理等职。在新浪广告、微博广告、西山居大数据平台架构中担任关键角色，并提供咨询服务。

随着 Red Hat Enterprise 7 的发布和 Ubuntu 系统的普及，systemd 开始逐渐进入到运维工程师的视线，从启动过程到服务管理，再到常用的计划任务、日志等日常运维工具都有 systemd 的影子，它对运维工程师的影响越来越大，这里就为大家介绍一下 systemd 在 Linux 日常管理中的使用。

### 6.1 systemd 和 sysVinit 之间的关系

systemd 是优秀的进程管理工具，在 CentOS 5.x 版本下管理进程的这类工具称为 sysVinit，管理员经常用的 service、chkconfig 等命令就是 sysVinit 提供的。但是 systemd 与 sysVinit 不同，它能够从系统的启动过程就开始进程的管控，对非终端依赖型的程序（服务）进行配置，而且还提供了很多进程通信和进程调用的简便用法；这里以 CentOS 的发行版本为例来为大家讲解，在启动过程中 systemd 和 sysVinit 的区别。

### 6.1.1 sysVinit 方式下系统的启动特点

运维工程师最熟悉的、也是目前应用最为广泛的生产环境的系统就是 CentOS 5.x 和 CentOS 6.x 了, sysVinit 是 Linux 操作系统上非常经典的系统初始化和进程管理工具, 甚至在实际在工作中, 很多熟练的运维工程师会使用 Shell 脚本来编写自己的服务启动脚本, 以方便应用程序在系统初始化时完成启动, 从而简化运维工作。但是我在长时间的使用中也发现了如下一些问题:

(1) 服务只能按照之前编排好的顺序来执行, 这会使启动过程变得缓慢, 而由于人为失误会导致某些服务在启动的时候“hang住”, 操作系统不能继续引导, 从而导致无法正常开机和关机, 我们希望能够通过简单的配置就可以绕过这种加载不正常的服务, 而不是用 Shell 来做各种异常的判断。

(2) 不能根据硬件状态的改变, 动态调整服务的启动和停止。无论硬件设备是否被激活, 所有的启动服务都要被加载, 不够灵活。

随着 Linux 的不断发展, 越来越多的人将 Linux 安装到个人设备上, 这时 sysVinit 在个人系统上的问题变得尤为突出, 因此另一个重要的桌面发行版 Ubuntu 操作系统的技术人员, 开发了 Upstart 方式来代替 sysVinit 方式。Upstart 支持分组顺序启动的方式, 即将有依赖关系的服务分到相同的组, 在组内按照顺序进行引导, 以达到并行启动的目的。但是这种方式依然不是启动速度最快的。

### 6.1.2 systemd 方式下系统的启动特点

为了进一步提升系统启动的速度, 以及方便对服务器进行管理, CentOS 7 采用了更先进的 systemd 来替代 sysVinit 和 Upstart, 而且管理命令和启动脚本的编写与 sysVinit 还可以兼容, 降低了运维工程师的学习成本, 所以之前在 CentOS 5、CentOS 6 上可以正常运行的服务启动脚本能够无缝迁移到 CentOS 7 上运行。

那么 systemd 有哪些新的特性呢? 其最大的特点就是可以真正地并行启动了。这种并行启动的方式和 Upstart<sup>①</sup>不同, 不需要用户来解决依赖关系, 例如: 在系统启动的过程中, 都会有依赖关系的问题, 像是 Apache、Nginx 这类的服务在启动之前要判断端口是否被占用, 这就需要获得连接到网络接口的 IP 地址, 如果网络服务没有启动, 就会导致依赖它的服务启动失败。在真正的启动过程中, 会出现更复杂的依赖, 即多层依赖, 如 A 服务依赖 B, B 服务依赖 C, 以至于更多的依赖关系, Upstart 启动的过程就会要求服务必须按照 C → B → A 的启动顺序才能正确运行。那么, 一旦 C 服务启动缓慢就会导致 B 和 A 这两个服务进入等待状态, 从而导致系统启动缓慢。而 systemd 可以让 A、B、C 三个服务并行启动起来, 而且还可以支持按需启动。

① Upstart 是一个基于事件的 init 守护进程的替代工具, 用于在系统启动时进行程序的管理。由 Canonical 公司的前雇员 Scott James Remnant 开发, 最初用于 Ubuntu 发行版。

那么，什么是按需启动呢？按需启动是指服务在没有被其他服务依赖时，是不会随着系统启动一起被加载的，这样就可以大大缩短启动的时间。这么“神奇”的功能是怎么实现的呢？首先要解决的就是依赖关系，systemd 作者总结出了三种服务依赖关系：

(1) 端口依赖，即服务 A 若正常启动，则需要先连接服务 B 的端口。解决这种依赖需要 systemd 先“制作”一个端口，当程序 A 启动时，systemd 检测到对该端口的 socket 请求，然后检测 B 服务是否启动了，如果没有启动则对 A 服务的请求进行缓存，直到程序 B 运行为止，这样就解决了端口依赖的问题。而且还很体贴地为用户提供了通过 systemd 拉起 B 服务的功能，进一步减少用户的工作量。

(2) 程序通信的依赖，systemd 采用了 D-Bus 作为程序之间的通信工具，在启动过程中，一旦被依赖的服务程序没有运行，同样会将请求进行缓存直到被依赖的程序运行为止。

(3) 文件系统依赖，除了“/”目录必须串行启动之外，其他被程序依赖的目录都可以采用和 autofs 相同的触发机制，即先创建临时挂载点（一个空目录），一旦有服务程序使用到这个挂载点则触发挂载的系统调用，将真正的目录挂载到事先定义好的挂载点，这就是 systemd 能进行并行启动的原因。

有了这三种解决依赖关系的方法之后，可以将启动过程中产生的依赖关系先缓存到内存中，然后继续启动下一个服务；systemd 就是这样让系统启动过程并行运行起来的，最后通过多 CPU 核心来并行处理，从而缩短了启动的时间。接下来我们来具体分析各种启动方式在系统底层都做了哪些工作。

## 6.2 systemd 的原理和启动顺序

### 6.2.1 sysVinit 的启动顺序

既然 systemd 由 sysVinit 进化而来，因此为了更好地掌握 systemd 的原理和启动过程，首先我们详细分析一下 sysVinit 的启动方式。目前我所在的公司还有很多业务运行在 CentOS 5 系统上面，该版本也是用于生产环境非常稳定的发行版本。下面我们以典型的 CentOS 5 系统为例，为大家介绍 sysVinit 体系下的启动顺序。

(1) 在 Linux 系统的启动过程中，首先由 BIOS 加电自检，之后将引导功能交给 GRUB 进行引导，用户可以根据界面提示来自由地选择不同版本的内核。

(2) 内核在引导完成之后，会加载第一个进程 init，这时就正式进入了 sysVinit 的引导环境，init 进程在磁盘的“/sbin/init”这个位置。之后所有的进程都是由它派生（fork）出来的，它的 PID 永远为 1。

(3) init 程序运行时会加载配置文件 /etc/inittab，在配置文件中根据指定的参数来确定自己的启动级别（runlevel），不同的启动级别可用来定义一组不同服务的启动顺序，在默认情况下会有 0 ~ 6 共 7 个启动级别，也就是说用户可以定义 7 种（实际上是 5 种）不同的启动

方式，以保证用户可以灵活地进行服务启动顺序的设置。默认情况下，在生产环境中我们将启动级别设置为 3，其他启动级别的用途将在 6.2.2 节为大家详细介绍。

(4) `init` 程序会按照默认的级别来逐个加载一组服务，首先会加载各个外设的硬件驱动，然后加载基础服务，如网络参数、蓝牙、软件 raid、LVM 逻辑等服务，之后再加载其他的应用服务。全部加载完成之后会进入等待用户登录的界面，来完成一次启动过程。

这就是 `sysVinit` 的启动过程，引导过程大量使用了 Shell 脚本，以方便运维工程师分析启动过程中都执行了哪些操作，这种简单又直接的启动方式很适合运维工程师去理解启动的过程，但启动效率却很低。

`sysVinit` 的启动过程如图 6-1 所示：

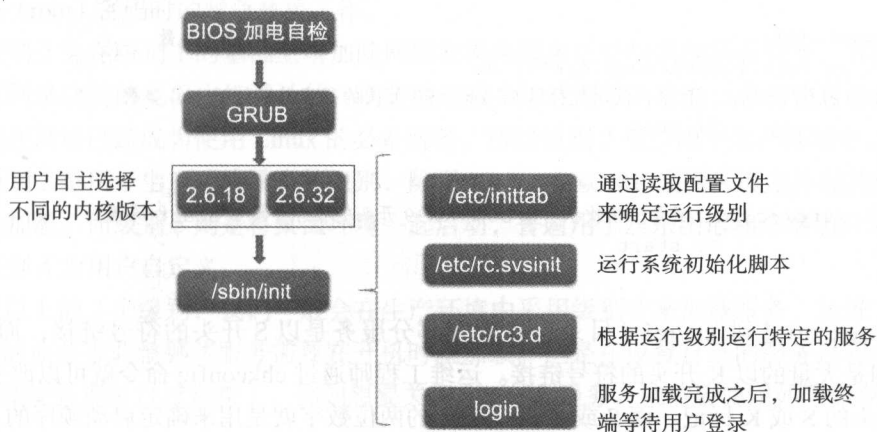


图 6-1 `sysVinit` 的启动过程

为了便于对比不同的启动程序之间的差异，下面将更具体地分析下 `/etc/inittab` 文件中的内容。

(1) `init` 进程加载 `/etc/inittab` 配置文件的部分，获得运行级别后并不会根据级别立即启动这组服务程序，`init` 进程会为系统执行初始化动作。这个初始化动作是由 `/etc/inittab` 的下列选项来完成的：

```
si::sysinit:/etc/rc.d/rc.sysinit
```

它的含义是在系统启动各个服务程序之前执行 `/etc/rc.d/rc.sysinit` 脚本，这个脚本是串行地将主机名、文件系统、swap、SELinux、udev、内核参数、系统时钟、RAID、LVM 等服务启动时可能会依赖的功能先运行起来，然后再加载指定级别的服务。如果我们测量一次完整的启动时间，会发现 `rc.sysinit` 脚本的运行时间占用了大部分的启动时间，因为它为后面的各级别服务准备了基础环境，而不关心后面的服务是否会用到这些资源，从而造成了大量的启动时间的浪费。

(2) 加载服务又是怎样实现的呢？可以通过设置 `/etc/inittab` 配置文件的 “`id:3:initdefault:`” 这一行语句来更改系统默认的运行级别。在配置文件中还有这样的内容：



```
10:0:wait:/etc/rc.d/rc 0
11:1:wait:/etc/rc.d/rc 1
12:2:wait:/etc/rc.d/rc 2
13:3:wait:/etc/rc.d/rc 3
14:4:wait:/etc/rc.d/rc 4
15:5:wait:/etc/rc.d/rc 5
16:6:wait:/etc/rc.d/rc 6
```

这里是一个分支，用于保证系统会加载到一组指定的服务，而不是加载全部的服务。而前面的 wait 标志，会让系统逐个加载服务直到加载完所有的服务后才继续下一步的启动。

根据 initdefault 给定的级别（这里是级别 3），执行相应的服务。系统会执行 /etc/rc.d/rc 3 的 Shell 脚本，下面将跟踪 /etc/rc.d/rc 这个脚本，主要逻辑如下：

```
runlevel="$1"                                # 获取命令的参数
for i in /etc/rc$runlevel.d/K* ; do
# 进入 /etc/rc3.d/ 目录，依次执行以 K00 ~ K99 开头的程序并增加 stop 参数
    $i stop
done
for i in /etc/rc$runlevel.d/S* ; do
# 进入 /etc/rc3.d/ 目录，依次执行以 S00 ~ S99 开头的程序并增加 start 参数
    $i start
done
```

打开 /etc/rc1.d 到 /etc/rc5.d 目录会发现大部分服务是以 S 开头的符号链接，而 rc0.d 和 rc6.d 下则是大量的以 K 开头的符号链接。运维工程师通过 chkconfig 命令就可以改变这些符号链接开头的 S 或 K 标记。而 S 或 K 标记后面的两位数字就是用来确定启动顺序的。

从这个逻辑我们不难发现，服务启动的过程是顺序完成的，而且在前一个服务没有执行完成之前，后面的程序都要等待；利用这种方式启动服务的好处还是简单，但这一过程会让现代计算机（多核并发）的机制产生严重的资源浪费，因为仅有单线程在工作。鉴于上面的这种顺序引导的情况，CentOS 6 采用了 Upstart 工具来代替 sysVinit 进行引导，Upstart 对 rc.sysinit 脚本做了大量的优化，它将启动需要做的每一个服务都封装成一个任务（job），并且为每一个任务增加一个状态，在启动时制定一些状态，并根据状态来确认每个任务运行到哪一步了，然后使用事件（event）机制和其他任务进行通信，这样就缩短了系统初始化时的启动时间。

有了 Upstart 之后，我们在同样的硬件环境下进行了启动时间的比较，发现启动一套标准的 CentOS 6 系统要比 CentOS 5 系统快很多。但是我们还发现如果在服务引导的部分还有可以优化的余地，即 rc3.d 部分引导服务的时候仍然是串行的，那么我们还可以对这部分缩短启动时间，接下来我们看看 systemd 是怎么做的。

## 6.2.2 systemd 的启动顺序

CentOS 5、CentOS 6 加载一组服务通常称作 runlevel（运行级别）。而在 systemd 启动的过程中，将使用 target（目标）取代 runlevel，这是启动过程中最大的变化。



默认的 runlevel 从 0 ~ 6 共 7 个级别，可以在运行时使用 runlevel 命令来查询当前的运行级别，通过 init 命令切换运行级别。

- 级别 0 和级别 6 分别表示关机和重启，它们都会调用 /etc/rc\$runlevel.d/ 目录下以 K 开头的服务，按照顺序调用 service < 服务名称 > stop 来关闭服务；唯一的区别就是级别 0 最后会调用 halt 命令关机，而级别 6 会调用 reboot 命令重启：

```
[user@localhost ~]$ runlevel
N 3
```

- 级别 1 会在系统启动时加载 /etc/rc1.d/S 开头的服务，这些服务只包含启动必需的服务，只会加载 bash 而不进行用户身份的校验，该级别可用于服务器异常或忘记管理员 (root) 密码时的紧急救援工作。
- 级别 2 会在级别 1 的基础上增加除网络和网络服务之外的其他所有服务，其最初是为无网络功能的桌面主机设计的，期望通过减少加载网络的功能，来加快启动速度。但现在网络已经成为使用 Linux 的必备所需，所以级别 2 很少用于生产环境中。
- 级别 3 是用于生产环境最多的级别，除了 KDE、GNOME 桌面环境之外的其他级别都会加载。而级别 5 则是将桌面环境一起启动，普遍用于娱乐图形和终端用户环境中。
- 级别 4 为用户自定义。

对于以上的 7 个级别，我们一般会在生产环境中采用级别 3 来加载服务，运维工程师会根据自己的业务来调整哪些服务需要在开机时就加载。但是在设置自己的服务过程中一旦有依赖关系就要管理员自行判断了，例如想要关闭 network 服务，就需要使用如下命令：

```
[user@localhost ~]$ chkconfig --level 3 network off
```

但其他依赖 network 服务的命令在下次启动的时候会失败，需要管理员根据经验自行判断，这点极易造成系统无法正常启动，而且灵活性也较低。

而在 systemd 中，target 默认有 40 多种组成方式，当系统启动时，会根据配置文件的关键字来确定依赖关系并进行加载。下面是 runlevel 和常用的 target 对比：

sysVinit 运行级别 (runlevel)	systemd 目标 (target)
0	runlevel0.target -> poweroff.target
1	runlevel1.target -> rescue.target
2	runlevel2.target -> multi-user.target
3	runlevel3.target -> multi-user.target
4	runlevel4.target -> multi-user.target
5	runlevel5.target -> graphical.target
6	runlevel6.target -> reboot.target

我们可以看到级别 0 和级别 6 使用的数字已经转换成了更好识别功能的英文，其中级别 1 为恢复模式；级别 2、3、4 合并成了一个 target，这个 target 被称作多用户模式，即不带图形的多用户网络模式；级别 5 单独使用一个 target 用于启动时启动 X 服务，即图形服务器。精简了基本的模式，并转化为英文之后，用户可以更好地区分每个 target 的功能；其他

的 target 可通过基本 target 文件中的依赖形成树形结构，继续进行引导。

所有的 target 都可以通过如下命令来查看：

```
[user@localhost ~]$ ls -l /lib/systemd/system/*.target
user.target
lrwxrwxrwx. 1 root root 17 Oct 29 17:20 /lib/systemd/system/runlevel3.target
-> multi-user.target
lrwxrwxrwx. 1 root root 17 Oct 29 17:20 /lib/systemd/system/runlevel4.target
-> multi-user.target
lrwxrwxrwx. 1 root root 16 Oct 29 17:20 /lib/systemd/system/runlevel5.target
-> graphical.target
lrwxrwxrwx. 1 root root 13 Oct 29 17:20 /lib/systemd/system/runlevel6.target
-> reboot.target
-rw-r--r--. 1 root root 402 Sep 15 21:21 /lib/systemd/system/shutdown.target
-rw-r--r--. 1 root root 402 Sep 15 21:21 /lib/systemd/system/multi-user.target
... ..
```

这里需要重点关注一个参数，它是涉及依赖关系的，打开这些 target 会发现它们使用了 Requires/Wants 关键字来确认各个 target 之间的依赖关系。其中 Requires 表示目标之间是强制依赖关系，Wants 表示目标之间是非强制依赖关系，运维工程师只需要关注需要启动的服务即可，而不再需要分析各个默认的 target 之间的依赖关系。那么依赖关系为什么会分为两种呢？通常用于来解决一种极端情况下的依赖关系——循环依赖，即  $A \rightarrow B \rightarrow C \rightarrow A$ ，出现这种循环依赖关系的时候，systemd 会尝试去掉 Wants 依赖，看能否正常进行引导。如果依然无法打破依赖关系，则会报错。

```
[root@localhost ~]# cat /lib/systemd/system/multi-user.target
```

```
[Unit]
Description=Multi-User System
Documentation=man:systemd.special(7)
Requires=basic.target
Conflicts=rescue.service rescue.target
After=basic.target rescue.service rescue.target
AllowIsolate=yes

[Install]
Alias=default.target
```

如果我们自己编写一个 target，那么怎样才能知道一个目标需要哪些服务呢？可使用如下命令：

```
systemctl show -p "Wants" multi-user.target
```

除了 Wants，还可以有其他各种形式的依赖和被依赖关键字，如 WantedBy、Requires、RequiredBy、Conflicts、ConflictedBy、Before、After 等。我们会在后面陆续用到这些依赖关系。介绍了这么多，接下来让我们看一个完整的目标配置文件的实例。

下面以 CentOS 7.0 系统自带的 NTP 服务为大家分析一下 systemd 引导的全过程。systemd 在内核加载完成之后，第一个引导的程序称为 systemd 命令。命令的绝对路径为 /usr/lib/systemd/systemd，它被作为 1 号进程，替代了 sysVinit 方式的 /sbin/init：

```
[root@localhost ~]# ps -ef | grep systemd
root 1 0 0 Dec02 ? 00:00:02 /usr/lib/systemd/systemd --switched-root --system
--deserialize 21
[root@localhost ~]# ls -l /usr/lib/systemd/systemd
-rwxr-xr-x. 1 root root 1230912 Sep 15 21:21 /usr/lib/systemd/systemd
```

systemd 已经使用目标替代了运行级别，所以在 /etc/inittab 中是看不到默认启动级别的。第一个目标会加载 /etc/systemd/system/default.target 配置文件；default.target 文件通常是符号链接，会链接到 /lib/systemd/system/<target name>.target，<target name>.target 我们通常会使用 multi-user.target 和 graphical.target，即 sysVinit 的级别 3 和级别 5：

```
[root@localhost ~]# ls -l /etc/systemd/system/default.target
lrwxrwxrwx. 1 root root 37 Oct 29 16:41 /etc/systemd/system/default.target -> /
lib/systemd/system/multi-user.target
```

如果需要更改默认的目标则可以使用 ln -sf /lib/systemd/system/<target name>.target /etc/systemd/system/default.target 命令，我们这里以 multi-user.target 为例，systemd 找到默认级别后会读取 multi-user.target，主要内容如下：

```
[Unit]
Requires=basic.target
After=basic.target rescue.service rescue.target
[Install]
Alias=default.target
```

这是一个标准的目标文件内容，格式上一般会有 [Unit] 和 [Install] 两部分，[Unit] 部分的 “Requires=” 表示强制依赖关系，即 multi-user.target 若要正常启动则需要先调用 basic.target；“After=” 表示要在 basic.target rescue.service rescue.target 这 3 个 target 之后才能引导，是起引导顺序的功能，我们还会在 [Unit] 里面见到 “Wants=” 的设置项，它和 Requires 一样都是表示依赖关系，但是会在依赖冲突的时候暂时解除依赖关系。

[Install] 关键字在这里表示开机是否需要加载，和 sysVinit 的 chkconfig on 设置开机时启动哪个服务器的功能是相同的。在这里第一个引导的 target 必须为 Alias=default.target 固定值才能继续加载服务。

继续跟踪引导过程，由于存在 Requires=basic.target 的设置项，因此 systemd 会读取 basic.target 是否还有依赖，直到所有的 Requires 都检测完，当依赖的 [Unit] 都被激活后，这个 [Unit] 才会被引导；如果有任何一个没有被激活，那么启动的时候会跳过这个 [Unit]。如果要自定义启动逻辑，除了启动时激活必要的 [Unit]，自定义的启动过程建议使用 Wants 关键字，以避免人为失误导致系统无法加载。

除了 target 文件本身可以使用 “Wants=” 关键字，还可以使用 <target 名字>.wants 文件夹，并在里面写入其他 target 或服务来表示依赖，例如在 /lib/systemd/system 下，不但有 multi-user.target 文件，还有 multi-user.target.wants/ 目录，里面还包含了其他 target 和 service 的符号链接。它们在我的测试机上面的格式如下：

```
[root@localhost system]# pwd
/lib/systemd/system
[root@localhost system]# ls -l multi-user.target.wants/
total 0
lrwxrwxrwx. 1 root root 16 Mar 24 16:06 brandbot.path -> ../brandbot.path
lrwxrwxrwx. 1 root root 15 Mar 24 15:34 dbus.service -> ../dbus.service
lrwxrwxrwx. 1 root root 15 Mar 24 16:05 getty.target -> ../getty.target
lrwxrwxrwx. 1 root root 24 Mar 24 15:35 plymouth-quit.service -> ../plymouth-quit.service
lrwxrwxrwx. 1 root root 29 Mar 24 15:35 plymouth-quit-wait.service -> ../plymouth-quit-wait.service
lrwxrwxrwx. 1 root root 33 Mar 24 16:05 systemd-ask-password-wall.path -> ../systemd-ask-password-wall.path
lrwxrwxrwx. 1 root root 25 Mar 24 16:05 systemd-logind.service -> ../systemd-logind.service
lrwxrwxrwx. 1 root root 32 Mar 24 16:05 systemd-user-sessions.service -> ../systemd-user-sessions.service
```

我们可以看到 multi-user.target 的依赖关系中除了其他的 target 还有 service，service 在这里称作服务单元配置文件，特点是以 .service 结尾，service 文件用来封装真正需要启动的服务。可以简单地将其理解为 sysVinit 时代的启动脚本（它们通常被放在 /etc/init.d 下的可执行文件中）。通常情况下 service 文件会隐含地包含以下 4 个设置：

- ❑ Requires=basic.target
- ❑ After=basic.target
- ❑ Conflicts=shutdown.target
- ❑ Before=shutdown.target

这 4 个设置用于保证该服务是在基础环境加载完成之后才开始加载的，在系统关机前正确的结束，然后将缓存数据刷新到磁盘中。如果手工启动或停止某个不存在的 service，那么 systemd 会在 /etc/init.d 中寻找同名的 sysVinit 脚本，并且根据同名脚本自动创建一个 service 单元，这就是之前我们提到的 systemd 兼容 sysVinit 脚本的方式了。

具体应该怎么引导呢？接下来我们看一个 .service 文件，这里是通过一个简单的 NTP 服务单元来为大家分析一下：

```
ntp.service
[Unit]
Description=Set time via NTP
After=syslog.target network.target nss-lookup.target
```

```
Before=time-sync.target
Wants=time-sync.target

[Service]
Type=oneshot
ExecStart=/usr/libexec/ntpddate-wrapper
RemainAfterExit=yes

[Install]
WantedBy=multi-user.target
```

首先我们来看 [Service] 部分，每个 .service 配置文件都必须包含 [Service] 部分，这个部分用于对允许程序的行为进行管理，其中 “Type=” 设置的是进程启动的类型，常见的类型包括 simple 和 oneshot 两种，当 Type=simple 的时候，“ExecStart=” 指定的进程就是该服务的主进程，会一直运行下去，直到 systemd 结束为止，或者运维工程师手动将此服务结束。而 oneshot 则表示 systemd 启动后续的服务之后这个程序会退出。在这个例子中 NTP 服务是 onshot 类型，因此在这个 servie 文件启动成功之后，“ExecStart=” 后面跟着的 ntpdate-wrapper 会退出，而 “RemainAfter=” 参数一般会作为 onshot 类型的补充，用来表示这个 service 下所有的执行文件都执行完退出之后，状态是否还是 active 状态。因为在使用一次性程序，特别是这种典型的网络程序的时候，程序执行一次后就会退出了，如果通过检查进程是否存在还是无法确定程序是否成功执行过，那么通过这个参数就可以确定，systemd 会保持这个 service 是 active 状态，以供其他的 service 进行检测。

另外，我们再介绍一种 Type 类型：Type=Dbus，它最典型的应用就是 NetworkManager.service，使用 Dbus 类型不但要指定类型，还要设置一个 BusName，后续启动的进程有依赖的时候，就会检测到这个 BusName，进而通过 socket 通信。这个隐含选项调用了 dbus.socket 文件的 [Socket] 来指定 socket 通信的位置。

从这个文件里面我们看到还有 [Unit] 小节，这里有两个关键字 “Before=” 和 “Wants=”，这两个关键字用于保证这个服务的加载顺序，要在 syslog.target network.target nss-lookup.target 加载完成之后才能加载，但不必考虑依赖关系，而 Before=time-sync.target 则要求它在 time-sync 这个 target 之前就加载。这就是 sytemd 解决引导顺序的方法。

systemd 就是利用这些关键字完成了复杂的启动依赖关系，运维工程师甚至可以在不懂 Shell 脚本的情况下就可以配置各个服务之间的依赖关系了，而且还是并行引导的。systemd 的引导过程就为大家介绍到这里了，接下来我们来看看 systemd 提供了哪些系统管理的命令，可用来方便我们进行运维工作。

## 6.3 systemd 的进程控制命令

systemd 兼容了 sysVinit 的进程控制类命令，同时还把需要修改配置文件才能生效的功

能也封装成了新的命令。这里介绍一下 systemd 新增加的 systemctl、hostnamectl、localectl、loginctl、timedatectl 命令。

6.3.1 systemctl 命令

sysVinit 环境下最常使用的命令是 service 和 chkconfig，chkconfig 命令可用于查看和设置开机自动加载的服务，service 可用于控制服务当前的运行状态。这里先来看下 chkconfig 命令和 service 命令的常用方法：

```
[root@localhost ~]# chkconfig --list network | grep network
network          0:off    1:off    2:on     3:on     4:on     5:on     6:off

[root@localhost ~]# service network help
Usage: /etc/init.d/network {start|stop|status|restart|reload|force-reload}
```

service 和 chkconfig 这两个命令可被 systemctl 替代，以 network 服务为例，来看下它们的对应关系，如表 6-1 所示：

表 6-1 systemd 与 sysVinit 服务管理命令对照表

systemd	sysVinit	功 能
systemctl start network.service	service network start	立即运行 network 服务
systemctl stop network.service	service network stop	立即停止 network 服务
systemctl enable network.service	chkconfig network on	开机加载 network 服务
systemctl disable network.service	chkconfig network off	开机不加载 network 服务

通过表 6-1 的对比，大家应该很容易理解 systemctl 命令，它将两个命令整合到了一起，因此更方便使用。

6.3.2 hostnamectl 命令

在 CentOS 5.x/6.x 的工作环境中，设置主机名往往需要多步操作才能完成，首先要通过 hostname 命令设置当前的主机名，为避免主机名失效还需要将主机名写入配置文件；通过 systemd 提供的 hostnamectl 命令可以只用一条命令就能完成上述操作，hostnamectl 命令的执行方法如下：

```
[root@localhost ~]# hostnamectl set-hostname host1
[root@localhost ~]# hostnamectl status
Static hostname: host1
Icon name: host1
Chassis: n/a
Machine ID: 9f9558a2a43914e8b079acbdbbe9804c7
Boot ID: 4084e70ebc8047d3ae7eb01f5a41dbdc
Virtualization: kvm
Operating System: CentOS Linux 7 (Core)
```



```
CPE OS Name: cpe:/o:centos:centos:7
Kernel: Linux 3.10.0-229.20.1.el7.x86_64
Architecture: x86_64
```

### 6.3.3 localectl 命令

修改字符集，这也是管理员在服务器初始化时经常要做的事情，通常使用 `echo $LANG` 的方式查看当前系统的语言和字符集，为持久化语言和字符集，通常设置 `/etc/locale.conf` 配置文件来确保下次启动时生效。systemd 提供了 `localectl` 命令来进行设置，通常的用法是：

```
localectl list-locales 查看系统支持的字符集
localectl set-locale LANG=< 字符集 > 设置字符集
```

例如将系统改为简体中文 GBK 字符集，可以使用如下命令：

```
[root@localhost ~]# localectl set-locale LANG=zh_CN.gbk
```

打开新的终端和重启之后再查询字符集会发现它已经变成新的状态了，当前终端如果想要立即生效，可以使用如下命令进行修改：

```
export LANG=zh_CN.gbk
```

### 6.3.4 loginctl 命令

`loginctl` 是一个会话状态控制命令，它能够列出用户的会话信息，帮助管理员定位用户使用命令的情况，`loginctl` 命令常用的参数有 `list-sessions` 和 `session-status`，`list-sessions` 可以查看当前系统的用户 `sessionID`，通过指定的 `session ID` 可以查看某一会话的详细信息。`loginctl` 命令的执行方法如下：

```
[root@localhost ~]# loginctl list-sessions
```

SESSION	UID	USER	SEAT
1	0	root	
3	0	root	
200	0	root	

```
3 sessions listed.
```

```
[root@localhost ~]# loginctl session-status 200
200 - root (0)
    Since: Thu 2015-12-10 17:47:15 CST; 38min ago
    Leader: 3596 (sshd)
    Remote: remote.kvm.vt
    Service: sshd; type tty; class user
    State: active
    Unit: session-200.scope
        └─ 3596 sshd: root@pts/2
        └─ 3600 -bash
```



```

└─ 4060 loginctl session-status 200
└─ 4061 less

```

### 6.3.5 timedatectl 命令

为了统一早期版本的 `date` `hwclock` 这些查看和设置时间的命令，`systemd` 增加了一个时间管理命令 `timedatectl`，使用 `status` 参数可以查看当前的系统时间。该命令的执行方法具体如下：

```

[root@localhost ~]# timedatectl status
Local time: Thu 2015-12-10 18:32:33 CST
Universal time: Thu 2015-12-10 10:32:33 UTC
RTC time: Thu 2015-12-10 10:32:32
Timezone: Asia/Shanghai (CST, +0800)
NTP enabled: n/a
NTP synchronized: no
RTC in local TZ: no
DST active: n/a

```

使用 `set-ntp` 参数可以开启或关闭 NTP 时间同步服务，开启时间同步的命令如下：

```

[root@localhost ~]# timedatectl set-ntp yes

```

## 6.4 systemd 服务管理

除了系统自带的软件之外，在实际应用中我们经常还会部署其他的开源软件，如果也需要使用 `systemd` 管理它们的启动，那就需要自行编写 `systemd` 的 `service` 文件，在这里我们以 Nginx 为例，从大家熟悉的 `sysVinit` 方式开始，来介绍如何编写自己的 `systemd` 服务脚本。

### 6.4.1 编写 Nginx 的 sysVinit 启动脚本

#### (1) 下载 Nginx:

```

[root@localhost ~]# wget http://nginx.org/download/nginx-1.9.4.tar.gz

```

#### (2) 安装:

```

[root@localhost ~]# tar xzf nginx-1.9.4.tar.gz
[root@localhost ~]# cd nginx-1.9.4/
[root@localhost ~]# ./configure --prefix=/usr/local/nginx
[root@localhost ~]# make
[root@localhost ~]# make install

```

#### (3) 编写启动脚本:

```

#!/bin/bash
# nginx.sh

```

```

# 设置Nginx在runlevel 为 3 或 5 时,随着系统一起启动,如果不需要开机启动,
# 则可以设置为 chkconfig: - 13 68 ; 这里的 13 和 68 分别表示开机启动和关机时
# Nginx 的顺序
# chkconfig: 35 13 68
# description: nginx server
# http://nginx.org/download/nginx-1.9.5.tar.gz
# 定义变量
nginx_bin="/usr/local/nginx/sbin/nginx"
nginx_prefix="/usr/local/nginx/"
nginx_pid="/usr/local/nginx/logs/nginx.pid"
# 加载系统函数
. /etc/rc.d/init.d/functions

# 进程是否正在运行,若运行则返回 0,若没有运行则返回 1
function nginx_is_running {
    local nginx_pid_number=$(cat $nginx_pid)
    # /proc 下有和进程 PID 相同名称的目录,即为该进程正在运行
    if [ -d /proc/$nginx_pid_number ]
    then
        echo 0
    else
        echo 1
    fi
}

# 启动函数
function start {
    if [ -f $nginx_pid ]; then
        [ $(nginx_is_running) = 0 ] && echo "nginx is running"
        exit 2
    else
        # 启动 Nginx
        ${nginx_bin} -p ${nginx_prefix}
    fi
}

# 停止函数
function stop {
    if [ $(nginx_is_running) = 1 ]; then
        echo "nginx is not run"
        exit 2
    else
        # 停止 Nginx
        ${nginx_bin} -p ${nginx_prefix} -s stop
    fi
}

# 重启函数
function restart {
    stop

```

```

    start
}

function status {
    if [ -f ${nginx_pid} ]; then
        [ $(nginx_is_running) = 0 ] && echo "nginx is running"
        exit 0
    fi
    echo "nginx is not run"
    exit 2
}

case "$1" in
    start)
        start
        ;;
    stop)
        stop
        ;;
    restart)
        restart
        ;;
    status)
        status
        ;;
    *)
        echo "usage: $0 {start|stop|restart|status}"
        ;;
esac

```

那么，应该怎样使用呢，我们将脚本复制到 /etc/init.d/ 目录下，并为其设置可执行权限：

```

[root@localhost ~]# cp nginx.sh /etc/init.d/nginx
[root@localhost ~]# chmod 755 /etc/init.d/nginx

```

然后使用命令：

```

[root@localhost ~]# chkconfig --add nginx

```

使 Nginx 开机能够自动运行。

sysVinit 脚本是在 Centos5.x/6.x 系统上面运行的，接下来我们看看 systemd 如何编写 Nginx 的启动脚本。

## 6.4.2 编写 Nginx 的 systemd 启动脚本

systemd 启动脚本不需要编写 Shell 脚本，这里建立一个 Nginx 的 systemd 启动脚本，脚本用 nginx.service 命名，nginx.service 文件的内容具体如下：

```

[Unit]
Description=nginx - high performance web server

```

```

Documentation=http://nginx.org/en/docs/
#After 参数设置项用来确认启动的顺序
After=network.target remote-fs.target nss-lookup.target

[Service]
Type=forking
PIDFile=/usr/local/nginx/logs/nginx.pid
# ExecStartPre 参数可确保 ExecStart 参数启动之前执行的命令，这里是在启动之前进行配置文件正确性的检测
ExecStartPre=/usr/local/nginx/sbin/nginx -t

# ExecStart 参数用来启动 Nginx 服务
ExecStart=/usr/local/nginx/sbin/nginx
# ExecReload 参数指定重新加载时执行的命令
ExecReload=/bin/kill -s HUP $MAINPID
# ExecStop 参数指定停止 Nginx 服务时执行的命令
ExecStop=/bin/kill -s QUIT $MAINPID
PrivateTmp=true

[Install]
WantedBy=multi-user.target

```

将脚本放在 `/usr/lib/systemd/system/` 目录下，使用如下命令可重新载入所有的 [Unit] 文件，确保我们添加进去的 [Unit] 被系统加载。

```
systemctl daemon-reload
```

使用 `systemctl start` 命令运行 Nginx 服务，命令如下：

```
[root@localhost ~]# systemctl start nginx.service
```

如果 Nginx 成功启动，则可以查看到如下信息：

```

[root@localhost ~]# systemctl status nginx.service
nginx.service - nginx - high performance web server
    Loaded: loaded (/usr/lib/systemd/system/nginx.service; disabled)
    Active: active (running)
    Docs: http://nginx.org/en/docs/
    Process: 21244 ExecStart=/usr/local/nginx/sbin/nginx (code=exited, status=0/SUCCESS)
    Process: 21242 ExecStartPre=/usr/local/nginx/sbin/nginx -t (code=exited, status=0/SUCCESS)
    Main PID: 21247 (nginx)
    CGroup: /system.slice/nginx.service
            └─ 21247 nginx: master process /usr/local/nginx/sbin/nginx
               └─ 21248 nginx: worker process

nginx[21242]: nginx: the configuration file /usr/local/nginx/conf/nginx.conf
syntax is ok
nginx[21242]: nginx: configuration file /usr/local/nginx/conf/nginx.conf test
is successful

```

```
systemd[1]: Started nginx - high performance web server.
```

使用 `systemctl stop` 可以停止服务，再使用 `systemctl status` 命令可以看到如下信息：

```
[root@localhost ~]# systemctl stop nginx.service
[root@localhost ~]# systemctl status nginx.service
nginx.service - nginx - high performance web server
   Loaded: loaded (/usr/lib/systemd/system/nginx.service; disabled)
   Active: inactive (dead)
     Docs: http://nginx.org/en/docs/
```

通过上述信息可以得知，Nginx 服务已被正确关闭。这就是 `systemd` 的启动脚本，相信大家根据 6.2 节所述的原理可以实现大多数开源软件的启动脚本。

### 6.4.3 systemd 的其他功能

`systemd` 的功能不仅限于以上几种，它还能代替很多系统的功能，例如：`socket` 激活进程、识别硬件设备、挂载磁盘、执行计划任务等。

#### 1. 使用 systemd 的 socket 激活进程

在 `systemd` 没有出现之前，系统管理员经常会编写一些 Shell 脚本，有些脚本需要跨主机通信，那么就需要使用类似于 `xinetd` 的方式来进行端口绑定，我们先来看下 `xinetd` 封装的 `rsync` 服务：

```
service rsync
{
    disable = yes
    socket_type      = stream
    wait            = no
    user            = root
    server           = /usr/bin/rsync
    log_on_failure += USERID
}
```

在这个配置文件中，`service` 参数后面的服务名称可用于区别其他的服务，这里用的是 `service rsync`，还要注意此名称需要和 `/etc/services` 配置文件的名称相对应，因为在 `/etc/services` 里面绑定了服务名称和对应的 TCP 或 UDP 端口。

```
[root@localhost ~]# grep rsync /etc/services
rsync          873/tcp      # rsync
```

下面继续来看看其他参数：

- ❑ `disable` 参数用来标记是否开机启动此项服务，这里设置为 `yes` 的时候表示开机不启动 `rsync` 服务。
- ❑ `socket_type` 为接受请求的类型，如果 TCP 协议为流类型，UDP 协议为数据报类型，那么这里使用 TCP 协议，所以类型使用 `stream` 方式。

- ❑ `wait` 表示请求是否会阻塞，如果希望并发处理用户的请求可以将此参数设置为 `no`。
- ❑ `user` 是 `xinetd` 调用 `rsync` 的用户权限。
- ❑ `server` 是最重要的参数，这里表示的是绑定的应用程序，网络请求会转发到这个应用程序。这里将转发给 `rsync` 以处理接收到的数据。
- ❑ `log_on_failure` 是将错误信息记录到日志，这里使用了 `+=USERID` 参数，将把用户信息一起记录进去。

配置完成后可使用如下命令启动 `sync` 服务：

```
[root@localhost ~]# chkconfig rsync on
[root@localhost ~]# service xinetd restart
```

使用 `rsync` 命令访问 873 端口即可连接到 `rsync` 服务，过程略显繁琐，使用 `systemd` 集成的 `socket` 功能就会简单很多。

`systemd` 的这个特性称为 `socket` 激活进程功能，但是在使用 `socket` 激活进程的时候需要注意，应用程序必须支持 `socket` 激活，而且能够处理 `socket` 请求，所以像 Apache Nginx 这种 `daemon` 类型的程序是无法支持的，一般应用于 Shell 或 `docker` 上。例如，编写一个接收用户 `socket` 的请求，然后根据请求的内容启动 Nginx 程序的脚本。

首先需要创建一个 `socket` 类型的 `[Unit]` 文件，命名为 `start-nginx.unit`，内容如下：

```
[Socket]
ListenStream=8080
[Install]
WantedBy=sockets.target
```

这里的 `ListenStream` 表示的是 8080 端口，一旦收到请求就会调用和它同名的 `servie` 文件，我们将这个文件写入 `/etc/systemd/system/start-nginx.unit`，然后再编写同名的 `service`，将下面的配置信息写入 `/etc/systemd/system/start-nginx.service` 文件：

```
[Unit]
Description=start nginx
[Service]
ExecStart=/usr/local/startnginx.sh
```

很简单，`ExecStart` 是指在这个 `service` 启动的时候会调用 `/usr/local/startnginx.sh` 脚本，在脚本中可以使用 `read` 类命令接收用户的输入请求，然后根据自己的需要调用 Nginx 启动脚本即可。

最后将这个 `socket` 激活：

```
systemctl enable start-nginx.socket
systemctl start start-nginx.socket
```

当使用者请求到 8080 端口的时候就会调用 Shell 脚本来启动 Nginx 了。

目前这个功能在 CentOS7 上面并不完善，如果需要在生产环境中使用类似的功能建议还

是使用 `supervisord` 等软件来代替。`socket` 激活进程要在 `systemd` 209 版本的时候才能体现出作用，目前的 CentOS7 和 Ubuntu LTS 还在使用 208 版本，从 209 版本开始 `systemd` 工具提供了一个 `systemd-socket-proxyd` 小工具，可以将 `socket` 代理给后端的应用，用法如下：

```
[service]
ExecStart=/lib/systemd/systemd-socket-proxyd 127.0.0.1:8090
```

将请求转发到本机的 8090 端口上，然后通过 [Unit] 段 `Requires/After` 调用其他的 service 来控制程序的启动和停止。完整的示例，大家可以参考官方文档的 `systemd` 通过 `socket` 调用 Docker 的经典示例。这里面使用了 `systemd-nspawn` 来产生新的进程，这对我们掌握自己编写 `systemd` 的 service 文件也很有帮助。

## 2. 使用 device 进行设备识别

在使用 `systemd` 的 device 功能之前，我们先来介绍一下什么是 `udev`，相信大家都用过 Windows 的设备管理器，当有新的硬件添加或删除的时候，资源管理器会自动识别硬件的改变。而这一切在 Linux（2.6 内核以上）上就是靠 `udev` 来实现的，它在底层依赖了一个 `uevent` 接口，接收硬件的添加和删除事件。这种机制在桌面系统下没有问题，但是到服务器的生产环境中如果识别多块网卡的时候顺序是随机的，识别多块硬盘的顺序也是随机的，那么在系统的管理上就会造成很多困扰，而 `udev` 会根据硬件的特有属性对硬件设备和名称进行绑定，从而让用户访问固定的硬件设备。

device 功能可以将 `udev` 的规则封装在 `.device` 文件中，并以设备对应的名称来命名 device 文件，例如，我们插入一个特定的 u 盘，将其命名为 `/dev/usb1`，就可以建立一个 `dev-usb1.device`。在配置文件里写标准的 `udev` 规则库就可以了。`dev-usb1.device` 文件的内容如下：

```
[root@localhost ~]# cat dev-usb1.device
ACTION=="add",BUS=="usb",SYSFS{serial}=="0013729925234704A0",SYSFS{manufacturer}=="SanDisk",KERNEL=="sd*",NAME="%k",SYMLINK+="usb1",RUN+="/bin/mount /dev/usb1 /mnt"
```

## 3. 使用 mount 代替 /etc/fstab 功能实现文件系统挂载

在 Linux 的启动过程中，能进行开机自动挂载的配置文件有很多，而且还有 `automount` 实现触发式挂载，`systemd` 为大家提供了一种新的方法，让大家可以在服务启动的过程中进行挂载，`systemd` 还可以设置启动顺序和基于 `socket` 等方式的激活，这样就实现了 `automount` 方式的触发式挂载。这里我们以 CentOS7 系统自带的将 `tmpfs` 挂载到 `/tmp` 目录为例，来看下挂载是如何实现的：

```
...
[Mount]
What=tmpfs
Where=/tmp
```



```
Type=tmpfs
Options=mode=1777
...
```

这里截取最关键的部分 [Mount] 段，它的写法是 mount 命令的变形，mount 命令一般用法如下：

```
mount [ -t 文件系统类型 ] [ -o 选项 ] <设备名称> [ 目录 ]
```

其中：

- -t -o 根据用户需要都可以被省略。
- What= 是设备或虚拟设备的名称。
- Where 是本地目录。
- Type= 是文件系统的类型，这里的 tmpfs 是指内存。
- Options= 是挂载的时候要用的选项。

systemd 的配置可以翻译成如下的 mount 命令：

```
[root@localhost ~]# mount -t tmpfs -o mode=1777 tmpfs /tmp
```

如果增加一个选项又该如何修改呢？例如增加一个参数限制最大内存只能使用 100MB：

```
[root@localhost ~]# mount -t tmpfs -o mode=1777,size=100M tmpfs /tmp
```

只需要修改 “Options=mode=1777,size=100M” 即可。这就是 systemd 的 mount 方式了，配合 [Unit] 部分的 Before/After 就可以让管理员按照自己的需要来挂载文件系统了。

#### 4. 使用 timer 替代 cron

cron 是管理员经常会用到的工具，是用于定时执行脚本和备份的重要工具，这里给大家介绍另外一种方法，那就是使用 systemd 的 timer 来代替 cron 进行计划任务。

首先编写一个简单的测试脚本，来验证程序是否能够正常运行：

```
#!/bin/bash
# test cron
date >> /tmp/cron_test.log
```

它的功能非常简单，是将执行脚本的时间记录到临时文件。方便我们来测试脚本是否成功执行了。接下来编写一个调用这个脚本的 service 文件 crontimer.service：

```
[Unit]
Description=testcron
[Service]
Type=simple
ExecStart=/tmp/timer.sh
```

Type 所代表的执行方式有如下两种：

- Simple 表示基本的执行方式，让 Shell 正常解析我们的脚本就可以了。

- ExecStart 是执行任务的时候调用的脚本。如果有多个任务可以编写多个 service 文件，那么有依赖关系的话还可以使用我们在前面介绍过的 Before/After 方式来进行顺序的编排。

继续编写 timer 文件 crontimer.timer:

```
[Unit]
Description=cron test
[Timer]
OnBootSec=10min
OnUnitActiveSec=1m
Unit=crontimer.service
[Install]
WantedBy=multi-user.target
```

其中:

- OnBootSec 表示在系统启动之后 10 分钟才开始运行。
- OnUnitActiveSec 是运行的周期，这里用做测试，所以每分钟运行一次。
- Unit 表示调用的是哪个 service，可通过上面的 service 来调用脚本文件。

最后让这个 timer 生效:

```
[root@localhost ~]# systemctl start crontimer.timer
[root@localhost ~]# systemctl start crontimer.service
[root@localhost ~]# systemctl start crontimer.timer
[root@localhost ~]# systemctl start crontimer.service
```

这些就是我们经常使用的运维工具在 systemd 中的实现了。

## 6.5 优化

我们可以看到 systemd 的功能已经非常强大了，除了系统自带的服务之外我们还可以增加自己的脚本，systemd 也为系统管理员提供了很多性能分析工具，可以根据分析结果来进行启动的优化。

### 6.5.1 使用 systemd-analyze 优化启动时间

系统的启动时间是管理员关注的最重要的一个标准，因此 systemd 提供了一个 systemd-analyze 命令，来分析启动时间，用到的第一个参数是 time，执行方法如下:

```
[root@localhost ~]# systemd-analyze time
Startup finished in 466ms (kernel) + 5.834s (initrd) + 40.483s (userspace) = 46.784s
```

通过命令的返回结果可以看到内核、initrd 和 userspace 三部分各自的启动时间，而 userspace 就包括了 systemd 加载各个服务的时间，那么具体是哪个应用程序启动耗时最长，

这里要用到第二个参数 **blame**，执行方法是：

```
[root@localhost ~]# systemd-analyze blame
12.166s firewalld.service
9.211s tuned.service
4.254s postfix.service
3.841s lvm2-pvscan@252:2.service
3.292s boot.mount
... ..
```

它会将启动耗时最长的服务（service）显示在最上面，我们根据这个提示就可以找到指定的服务来优化它。

另一种优化是根据依赖关系和启动顺序进行优化，这种优化需要观察全局的启动顺序和依赖关系。systemd 不仅提供了这样的分析功能，还提供了图形化的跟踪，使用的命令如下：

```
[root@localhost ~]# systemd-analyze plot > bootlog.svg
```

显示的结果如图 6-2 所示，非常便于管理员分析启动过程来优化启动时间。

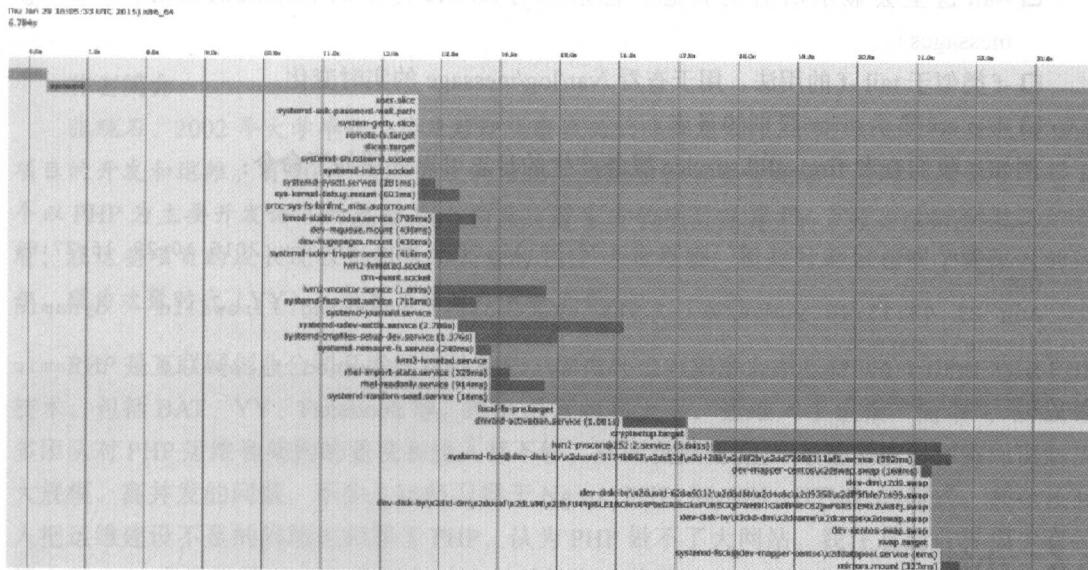


图 6-2 服务启动时间

除了要关注启动的速度之外，我们还会关注哪些服务在启动的时候没有按照预想的方式启动起来，这里可以使用 **systemctl-failed** 命令，使用方法如下：

```
[root@localhost ~]# systemctl --failed
UNIT                                LOADACTIVE SUB    DESCRIPTION
rngd.service loaded failed failed Hardware RNG Entropy Gatherer Daemon

LOAD    = Reflects whether the unit definition was properly loaded.
```

ACTIVE = The high-level unit activation state, i.e. generalization of SUB.  
 SUB = The low-level unit activation state, values depend on unit type.

1 loaded units listed. Pass --all to see loaded but inactive units, too.  
 To show all installed unit files use 'systemctl list-unit-files'.

该命令列举了启动失败的服务，可供管理员进行排错。当然，用户也可以使用 `systemctl status <服务名称>`，用于确定单个服务的运行状态。

## 6.5.2 使用 systemd journal 功能

systemd 还提供了日志查看和筛选的功能，我们可以利用此功能方便地进行日志查看，在没有使用 systemd 之前管理员大多是使用 `cat`、`more` 或 `tail` 配合管道使用 `grep` 进行过滤的，现在有了 systemd journal 的功能，就可以简化查询日志的命令了。

下面来看看有哪些常用的参数：

□ `journalctl <参数>`。

□ `--all` 这里会显示所有的日志，包括所有 service 打印的和系统的日志。(`/var/log/messages`)。

□ `-f` 类似于 `tail -f` 的用法，用于查看 `/var/log/message` 的实时变化。

□ `-b-p err` 显示 error 级别的日志。

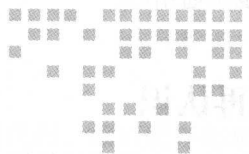
例如：我们查看 `firewalld.service` 服务产生的日志可以使用如下命令：

```
[root@localhost ~]# journalctl -u firewalld.service
-- Logs begin at Fri 2015-08-21 08:33:15 CST, end at Thu 2015-10-29 16:27:01
CST. --
Aug 21 08:33:40 yumserver-2.yhs systemd[1]: Starting firewalld - dynamic
firewall daemon...
Aug 21 08:33:52 yumserver-2.yhs systemd[1]: Started firewalld - dynamic
firewall daemon.
```

可以查看指定的服务产生的日志。

## 6.6 小结

systemd 是现代 Linux 操作系统上的非常实用的工具，通过上面的基本功能可以简化运维工程师大量的重复工作，但是 systemd 在使用习惯和用法上的改进也给运维工程师带来了巨大的挑战，希望这里能够通过对 systemd 的引导过程分析和特性的讲解，为大家掌握 systemd 的高级功提供帮助。



## 第 7 章 Chapter 7

# PHP 运维实践

### 作者简介

张观石，2002 年大学毕业，毕业后即从事桌面软件开发设计工作，后转型于互联网 Web 项目的开发和运维。有 10 年 PHP 开发和网站运维经验，曾负责运维“英雄联盟盒子”等多个以 PHP 为主要开发语言的大型 Web 项目，所负责的项目从日 PV 百万、千万到数亿的都有，在这些项目的成长过程中积累了大量的 PHP 运维经验，因此对运维也有了更深入的认识。现为欢聚时代 (YY) 互娱事业部业务运维负责人。

PHP 是互联网创业公司开发网站的首选，国内外也有很多大型互联网公司正在使用 PHP 技术，包括 BAT、YY、Facebook 等。PHP 语言容易上手，运维入手也很“简单”，所以很多团队对 PHP 运维和架构的重视和投入都不够，从而导致很多 PHP 技术栈的团队无法解决大规模、高并发的问题。不少人已经习惯于 Nginx+PHP+MySQL+HTML 的模式，甚至还有人把运维建设不足的问题也归罪于 PHP，认为 PHP 做不了大网站。我注意到业界很少有关于大型 PHP 项目运维技术的分享和讨论，于是尝试着借助本书分享一下自己的一些思考和实践。

本章首先会从开发和运维的多个新视角来重新认识 PHP，再讲解 PHP 运维与架构设计过程中的常见问题和解决方法，然后用 3 个小节的内容来分享运维最常见的工作：部署、性能分析和故障处理。先讲解 PHP 部署包括进程部署、配置变更及 PHP 代码发布；然后讲解如何分析 PHP 的性能问题、如何加强 PHP 的监控、如何调优 PHP；最后讲解如何快速定位和处理故障。本章中我们会说到业界的运维现状，会重点介绍我对 PHP 运维的思考和实践。本章的每一节都会介绍一些 PHP 运维实践，其中部分内容不仅仅局限于 PHP，对于其他技

术的运维工作也是通用的。

## 7.1 PHP 再认识

PHP 是网站技术中最流行的语言，权威语言排行网站 TIOBE 上显示全球有 83% 的网站使用了 PHP 技术。那么 PHP 到底是什么样的技术呢，接下来我们会试着从几个全新的视角来重新解析 PHP，以帮助读者更深入地理解 PHP，为提升 PHP 运维能力打好基础。

### 7.1.1 PHP 进程的工作方式

我们首先来了解下 PHP 是如何工作的，PHP 作为应用服务器时，目前普遍使用的是多进程工作模式，Web 服务器 Apache/Nginx 通过 FastCGI 协议把请求转发到 PHP-FPM 进程。下面就来分析一个 Web 请求生命的全过程（如图 7-1 所示）。

假设用户在浏览器地址栏输入 `http://www.test.com/index.php` 发起一个请求，然后：

❑ 域名被 DNS 解析到 Nginx 管理进程（Master process）所在的服务器 IP。

❑ Nginx 管理进程选择一个工作进程（Worker process）。

❑ Nginx 工作进程把请求转发到 PHP-FPM 管理进程（默认是 9000 端口）。

❑ PHP-FPM 管理进程分配一个工作进程处理 `index.php` 请求。

❑ 工作进程在服务器路径中找到 `index.php` 文件，解析编译。

❑ 执行 PHP 代码，可能还要请求后端存储等。

❑ 得到请求的结果，先返回给 Nginx，然后再返回给用户浏览器。

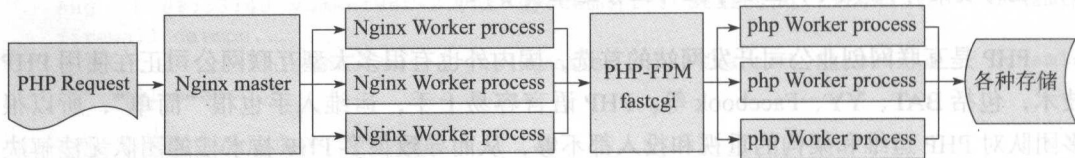


图 7-1 HTTP 请求在服务器上请求过程

PHP-FPM 管理进程不仅要负责分配 PHP 请求给工作进程，同时也要控制工作进程的创建、结束和启停。单个 PHP 工作进程服务完若干个请求后会结束进程，释放资源，管理进程再启动新的工作进程。PHP 多进程模式中内存等资源管理将由工作进程自行分配，满足一定的条件后重启工作进程会自动释放内存，即使内存泄漏也不会造成严重的问题，也不会出现多线程死锁的问题。所以 PHP 的可靠性较高，系统运行也更稳定，多进程需要不断地分配和回收进程资源，且需要消耗比线程模式更多的资源。多进程在大规模集群下的可扩展性很好，只需要简单地增加机器或增加进程即可实现扩展。

对于运维人员来说，多进程模式还有一个极大的好处，如果修改了配置文件则只需要重启工作进程即可，这也是为什么 Nginx、PHP-FPM 支持平滑重启，而 Tomcat、MySQL 等多



线程模式不支持平滑重启的缘故。

7.1.2 PHP 代码的编译和部署

PHP 属于半解释型语言，程序源代码先编译为中间代码，然后缓存起来，再由解释器读取缓存并逐行解释执行。PHP 的中间代码称为 Opcode，是介于 PHP 源码和机器代码中间的一种代码，只有 Zend 的执行器能读懂它。PHP 源码在首次执行时先编译为 Opcode 然后保存于内存中，下次将直接从内存读取 Opcode 并执行。PHP 7.X 也在大力推广内置的 Zend Opcode 扩展，以替换独立的 APC 扩展。JAVA 会被编译为 class 文件，发布出去，然后再执行，这就是所谓的一次编译，到处执行，而 PHP 是一次编译，多次运行。多次运行是指只要原文件不被修改，不刷新 Opcode，则内存中的 Opcode 缓存一直有效，这样就减少了从磁盘读取文件、词法、语法分析、编译的过程时间，从而提升了网站的性能。

PHP 的编译执行过程（如图 7-2）具体如下。

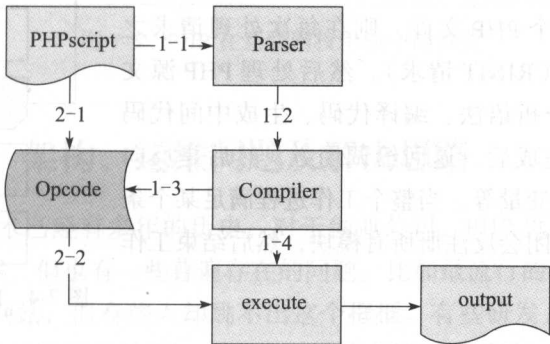
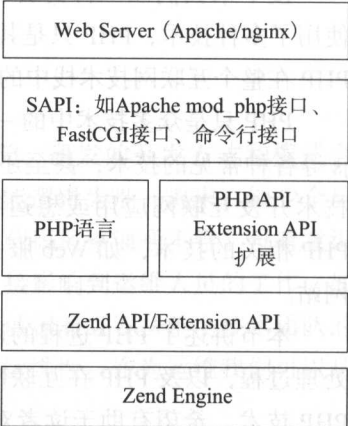


图 7-2 PHP 首次编译执行和从缓存执行的过程

第一次是执行语法词法分析和编译过程（如图 7-2 中的 1-1），编译后会保存为 Opcode（如图 7-2 中的 1-3），第二次直接从 Opcode 执行（如图 7-2 中 2-1），不再需要词法、语法和编译过程。



7.1.3 PHP 内部实现和生命周期

下面我们换个视角来理解 PHP 的实现，首先来看看“PHP”的组成部分。我们常说的 PHP 是指一门语言，其实 PHP 包括一系列相关的技术。

PHP 内部组成如图 7-3 所示，上层是 SAPI，它是连接 Web Server 的接口，可以服务于各种 Web 服务器，如 Apache，

图 7-3 PHP 的内部架构和组成



Nginx；第二层是 PHP 语言本身和 PHP 的扩展和函数的实现等；第三层是 Zend API，它把 Zend 引擎的指令封装成为 API 以供 PHP 语法层和扩展调用。整个架构的最底层是 Zend 引擎，他提供了 API 给上层使用，来实现 PHP API、PHP 语言和扩展。所以整个 PHP 技术最核心的就是 Zend 引擎，而“PHP 语言”只是 PHP 技术中不算很大的一块。

我们再来看看 PHP 内部是如何处理请求的，以及我们所熟悉的 php.ini 配置项又是如何配合执行的（如图 7-4）。了解内部工作过程有利于我们理解 PHP 配置和管理扩展。工作进程启动后首先会初始化各种全局变量和常量，初始化 Zend 引擎和核心组件，然后分析和应用 php.ini 文件的配置；启动核心后再依次注册各个模块（PHP 内部的模块化做得不错），模块“注册”完成后就可以开始处理 PHP 请求了。在处理请求的过程中会“激活”注册的所有模块，如果是通过 URL 请求某个 PHP 文件，则在每次处理请求之前都会进行模块激活（RINIT 请求）。然后处理 PHP 源文件，包括解析词法、分析语法、编译代码、生成中间代码 Opcode、执行。处理完成后，返回给调用方，然后是反向操作，停用模块、释放变量等。当整个工作进程满足某个条件时就会关闭，正常关闭会反注册所有模块，然后结束工作进程。

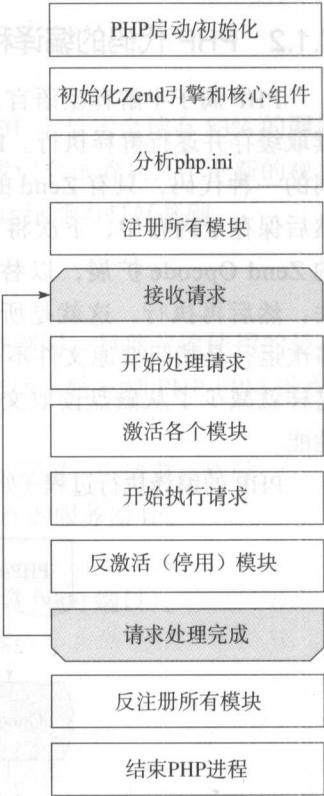


图 7-4 PHP 进程的内部处理过程

7.1.4 PHP 在互联网技术栈的位置

接下来我们以一种更宏观的视野来了解 PHP。PHP 是应用层的技术，而大多数网站都使用了多种技术，PHP 只是其中的一块。我们列举常见的技术（如图 7-5），可以很容易就能 PHP 在整个互联网技术栈中的位置。

PHP 只是众多技术中的一个，后面还有 Memcached、Redis、MongoDB、MySQL、Node.js 等各种常见的技术，甚至还有队列服务、Java 服务、分布式数据库、文件存储等。用 PHP 技术开发互联网应用或想运维好一个 PHP 互联网应用，都必须要先掌握整个技术栈中与 PHP 相关的技术，如 Web 服务器、MySQL 等各种存储、缓存技术，才能完全掌控住你的网站。

本节讲述了 PHP 进程的运行模式、请求在进程间传递的流程、代码在 PHP 进程内部的处理过程，以及 PHP 在互联网技术栈中的位置，从宏观和微观的多个不同视角重新认识了 PHP 技术。希望有助于读者对 PHP 有更深入的了解，从而有助于我们更好地架构和运维好一个 PHP 技术的网站。

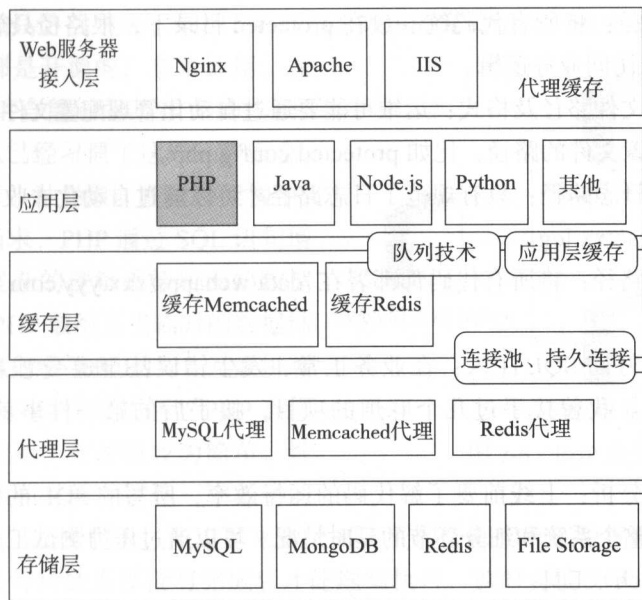


图 7-5 PHP 在互联网技术栈中的位置

## 7.2 PHP 开发、架构、运维问题及解决思路

业界使用 PHP 技术已经有多年的历史，对于创业公司，PHP 更是首选。业界已经形成了一些良好的最佳实践，但也有一些普遍存在的问题。比如最流行的 LNMP 架构，帮助了很多初创公司快速上线网站，但有些人却跳不出这个框框。有些研发人员喜欢用大型 PHP 框架，有些人喜欢用自己的微型函数库、有些人会做较全的架构设计，还有些人喜欢从头到尾流水模式执行下来。本节我们将讨论在 PHP 网站开发、架构设计、运维过程中运维的角色及可以发挥的作用。

### 7.2.1 运维对 PHP 研发提要求

PHP 代码不只是跟研发相关，跟运维工作也是密切相关的，研发的开发方式和模式会直接影响业务上线后的运维工作。有些开发人员以实现业务功能逻辑为唯一追求，缺少全局观，套个 HTML 页面完成业务功能就完事，代码效率、性能、代码风格通通不管。没有意识到 PHP 写得好坏也会直接影响到项目在线的运营质量，还会直接影响到运维人员的工作。如果 PHP 开发团队有多个小组，每个小组负责不同的业务，那么就很有必要从整个技术团队的范围来做一些统一规范，统一大家的开发风格，提升网站的可运维性。作为运维我们可以从自身角度提出一些具体的要求。

□ 统一的代码风格：MVC 模式组织代码、驼峰风格 / 下划线风格。

- 统一文件路径：将所有代码统一放在 `protected` 目录下，根路径只放 `index.php`，通过 URL rewrite 访问业务逻辑。
- 统一的配置文件路径及格式：运维可能要通过自动化管理配置文件，或者在排除故障时要修改配置文件的路径，比如 `protected/config.php`。
- 统一的应用日志路径：只有规范了日志路径才可以通过自动化去收集或清理日志，如 `protected/logs/xxx.log`。
- 统一的部署路径：将所有代码都部署在 `/data/webapps/xxx.yyy.com` 中，以业务域名作为部署路径。
- 要求代码做好防 SQL 注入：在业务正常、发生错误甚至遭受恶意输入时都不能有出错的输出。我曾接手过几个麻烦的项目，接手后的第一件事就是修复 SQL 注入漏洞。
- 性能和效率分析：上线前要了解代码的运行效率，所写的 SQL 的效率，整个系统的吞吐能力，整个系统和细分环节的延时情况。可以通过压力测试工具来分析，比如 `ab` (Apache Bench) 工具。
- 容错容灾分析：一旦出现故障，要有一定的容错容灾能力，比如说做到无状态化，`session` 和产生的内容不存储在本机中，当然机器出现故障或挂掉的时候要能够快速切换到其他服务器，在业务负载较大时要能够快速扩容。
- 较大型的团队开发一定要用框架，在选择框架时要充分考虑到性能问题和运维方法。

PHP 运维人员可以不懂怎么写 PHP 代码，但需要了解 PHP 代码是怎么工作的。优秀的研发和运维人员要有全局的视野，要了解 HTTP 协议、浏览器、DNS、CDN、前端反向代理、后端缓存、MySQL 等数据库存储、文件存储等组件特性及其与 PHP 代码之间的关系。

下面列举两个我遇到过的反面例子。

**【案例 1】** 有研发使用 `curl_exec` 函数访问外部接口，却不控制超时时间，当外部接口不稳定时，工作进程一直在等待对方返回，直到对方返回超时时间才继续执行。这时如果有多个这样的请求，那么所有的 PHP 进程都会被阻塞住，没有工作进程来处理其他请求。

**【案例 2】** 有些开发人员认为缓存与自己无关，是架构师或运维维护的责任。还有些开发人员不了解缓存的特性，认为只要能往里面写数据、读数据就行了，结果造成内存利用不合理的问题，而且也没起到缓存的作用。某程序员还把大量的一次性数据写入缓存，他不了解一次性数据下次是不会再读写的，没必要缓存，缓存一次性数据反而会让内存无法释放，他还找运维说这缓存有问题，内存不够要扩容。

## 7.2.2 运维参与 PHP 项目架构设计

PHP 之所以这么流行，离不开 LNMP/LAMP (Linux+Nginx/Apache+PHP+MySQL，后面

所讲的内容均以 LNMP 为代表,如图 7-6 所示)架构的完美组合。这个组合开发运维的效率很高,所有软件都是开源的,拿来即用。在项目的早期这个组合能够发挥很大的作用,时间久了整个团队已经习惯于这样一种组合,在架构上反而成了一种局限。在这种架构中,Web 服务器收到请求,PHP 通过 SQL 语句增删查改数据,把复杂的逻辑丢给后面的数据库(SQL 语句),PHP 数组充当临时的数据结构,获取到数据后把结果和 HTML 元素拼装在一起展示给用户。这种做法是把 PHP 当作一门胶水语言来使用,在逻辑较为简单、性能要求不是那么高的时候,这是一个不错的方式。但是随着项目越来越大,流量越来越大时,负载全都压给了数据库,性能问题就会出现,PHP 往往要等待数据库读写完成后才能继续执行,这时可能会认为是 MySQL 出现了性能瓶颈,进而认为是 MySQL 服务器不行,以为增加内存,更换更好的服务器就能解决这个问题。其实这是整个网站的架构出现了问题,是要重新考虑整个网站架构的时候了,运维要和研发一起讨论可能存在的性能瓶颈,找到解决方法,比如加缓存、静态化等。

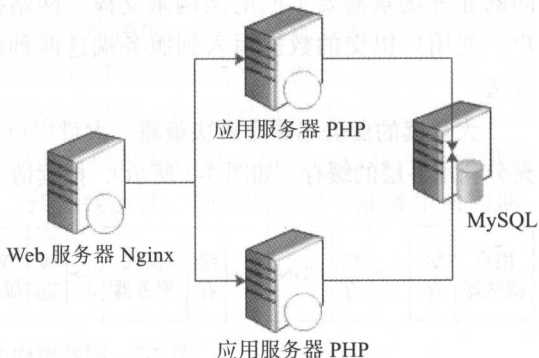


图 7-6 PHP 常见简单架构

解决之道:重新思考大型网站技术架构

网站架构不只是与 PHP 有关,而是要考虑全局的问题。目前互联网技术的趋势是分布式、缓存化(所谓的把内存当缓存用、把 SSD 当存储用、把磁盘当冷备用)、异构化、微服务化,对于架构和技术选型要有宽广的视野。

限于篇幅,这里只将总体思路总结如下:

(1) 理解整个请求的生命周期及 PHP 的位置,PHP 只是处理数据的一环,不能纯当胶水技术,而把逻辑都给了数据库;也不能完全依赖 MySQL 处理所有数据,在 PHP 层可适当地使用算法和适当的数据结构、缓存等技术来减轻数据库的压力。

(2) 对于实时性要求不是很高的页面应尽量静态化,以减少对 PHP 和 MySQL 的请求。

(3) 能缓存就缓存,包括浏览器缓存、CDN 缓存、反代缓存、Web 缓存、数据缓存比如 Memcache、Redis、Opcode 缓存、Mysql 缓存等,每一层缓存技术都值得大家深入研究。

(4) 需要用户等待较长时间的重型操作则用队列和异步操作来实现,将结果快速返回给用户。

(5) 若数据库达到瓶颈则要考虑做优化或拆分,横向分库分表、纵向按时间分表、按 ID 分表、读写分离等。

(6) 复杂逻辑比如特别消耗 CPU 和 IO 资源的,可在应用层使用其他技术来实现,比如 JAVA、C++、Go、Node、js、不要局限于 PHP,应合理利用各种技术的优势。

(7) 微服务架构,把重型服务拆分为微服务,进行面向服务的架构设计(SOA)。

PHP 的开发、运维和架构都与业务结合得很紧密，现在还没有一招能够通吃的架构，不同的业务场景需要不同的架构来支撑。网站技术的本质是把存储在服务端的信息展示给用户，把用户提交的数据写入到服务端这两种数据流向。下面来列举一些常见场景下的架构方法。

**大量读的业务场景之解决策略：**大量用户从服务器端读取数据的业务场景，架构设计应该充分利用各层的缓存（如图 7-7 所示），把读请求的数据尽量推到前面，以减少后端的读压力。

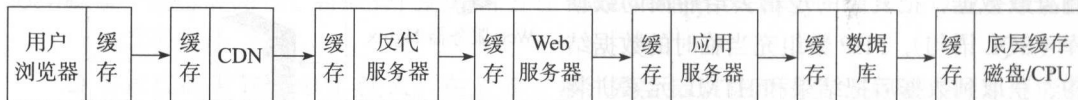


图 7-7 网站架构中各层缓存的效果图

比如用浏览器缓存使得部分资源不需要连接到服务器，利用 CDN 可以把读请求放在最后一公里，利用反向代理缓存，使得只需要读取到前端反代等。利用更多逻辑层的缓存，而不需要读取数据库。

**【案例 1】秒杀业务场景：**秒杀业务是一种比较特殊，但又经常会用到的场景。秒杀本质上是大量用户在同一时间去访问业务，判断产品是否还有剩余，如果还有就购买，没有了就返回失败。这种场景是典型的读多写少的场景，大量用户查询（读）是否有剩余商品，只有极少数商品可以进行下一步更新（写）状态，极端情况下同时有几十万次读操作而只有几次写操作。按照上面所说的策略是把读取和判断往前推，比如在 JS 阶段就判断时间是否在秒杀开启的范围内，如果超过开始时间多少秒就直接返回失败，把剩余商品数量存在内存缓存中，对超过并发的用户直接返回读取失败数据等。

**【案例 2】大量写的业务场景：**某客户端开启自动上传游戏战绩的功能，游戏拥有百万的同时在线用户，游戏结束后把 10 个玩家的比赛信息上传到服务器，经过分析、处理和统计玩家的游戏细节数据后存入数据库。每个上传文件 3MB 左右，分析处理比较耗时。架构思路：首先考虑去重，客户端去重，若 10 个玩家的信息是一样的（可通过文件 MD5 码是否一样进行判断）那么第一个玩家上传后其他九个人就不用再传了，最大可以减少 90% 的上传量；客户端也有可能上传失败，所以要保证有重试和延时上传的机会；先把上传的原始比赛详情数据包存入内存，同时写队列，把耗时的解析过程异步化，消费者读队列时再去解压原始数据进行分析入库；数据冷热分离，要将比赛战绩保留 1 个月，用 Redis 来保存，将过期数据存入冷库，MySQL 可做长期数据，比如用户数据、排名数据等。

### 7.2.3 PHP 运维常见问题及解决之道

PHP 的运维能力很容易被忽视，不信的话可以搜索 PHP 运维，看看有多少资料系统地讲解了 PHP 的运维工作。本书的读者大部分都是运维人员，更不能忽视 PHP 的运维，这也是本章的写作初衷所在。下面我们来说一说 PHP 运维常见的问题及解决之道。



如果有人问我们，PHP 运维是做什么的？或许回答是这样的：首先安装 PHP 软件包，再修改配置文件，然后启动 PHP-FPM，上传 PHP 代码。这种回答看起来也没什么错，然而并没有体现运维的价值。我个人很认同 Google 某大牛所说的，“运维是保证网站的可持续性运营”，在腾讯运维岗位被称为技术运营。在我的理解中运维是把产品设计、研发开发、测试完成的代码部署到服务器上，并保证以良好的状态持续运行，监控随时发生的状况，出了问题能够快速定位并解决，甚至进一步让运行数据反向推动产品、研发、运营人员改进业务。概括地说运维是服务于从立项到研发完成、到上线运营、到下线的整个业务生命周期。从中可以粗略地看出运维应该做的事情包括如下几项。

- 部署：包括基础设施的资源部署、软件环境的部署。
- 业务上线：PHP 代码发布。
- 稳定运行：保障用户进入网站页面后能一直顺利地完成整个业务流程。
- 监控：监控网站运行过程中的各种状态，如发生异常则告警。
- 性能优化：网站出现性能问题时能进行运维侧的优化，可帮助研发优化代码性能，容量不够时可快速扩容。
- 故障处理：出现故障后能快速定位到问题点。
- 数据分析反馈：对线上运行的状态能进行充分分析，把有用的结果反馈给研发和产品，推动产品进行下一次迭代。

运维要做的事情其实很多，每一个点都能体现出强大的运维能力。接下来几节将展开来讲其中的进程部署、代码发布、性能优化和故障处理这几个方面。

## 7.3 PHP 进程部署和配置、代码发布

在一台服务器中进行 PHP 进程的部署、配置和代码发布都很简单，只要安装 PHP 进程，对配置文件按需要进行修改，启动进程，把 PHP 代码复制上去即可，稍有经验的人都能完成这些工作。如何在较短时间内完成多台、十几台、甚至几十、上百台服务器的部署，而且不出差错，其实是很有挑战性的工作。网上有大量关于 PHP 部署的教程，不难发现网上教程的方法有介绍单台服务器 PHP 部署的，有讲 PHP 配置参数的，却很少有讲“大规模集群”下 PHP 的部署和配置变更的，讲 PHP 代码发布的也极少。本节将为大家分享一下我个人的一些思考和实践，PHP 部署可以包括以下三个内容：

- (1) PHP 进程的部署、升级，包括 PHP 扩展的部署。
- (2) PHP 配置文件的部署和变更。
- (3) PHP 应用代码的发布。

### 7.3.1 PHP 进程的部署

首先来看看 PHP 进程的部署，常见的两种部署方式包括源码编译和官方软件包管理，

然后介绍下互联网公司采用较多的自建包部署系统的方式，这里将重点介绍系统的思路。因为自建包部署系统不是每个公司都能做的，因此我们在最后分析出几个核心点，简述用批量运维工具来实现包部署的方法。

### 1. 源码编译

若要采用这种方式首先要下载 PHP 的源码，执行 `configure`、`make`、`make install` 三板斧，然后修改配置参数，启动进程。

源码编译可以更好地利用平台的特性，更灵活地指定一些优化参数，灵活地选择组件、扩展，可以编译、运行多个软件版本，但是会存在如下一些问题：

- ❑ 需要把源代码下载到每台服务器，时间长，效率低。
- ❑ 编译比较复杂，需要掌握更多的技术，如要解决包之间的各种依赖。
- ❑ 编译速度慢，时间长。
- ❑ 若优化不好性能可能会更差。

也有可能因为服务器 Linux 稍微有点不一致（版本差异或人为配置差异），编译过程和结果会很不一样，甚至编译通不过。所以对服务器进行源码编译的方式不适合大规模部署。

### 2. 软件包管理

每个 Linux 发行版都有各自的包安装方式和工具，官方会提供包的仓库。比如 RedHat 有 RPM 和 Debian、Ubuntu 有 `apt-get`、CentOS 有 YUM。通过 Linux 官方包管理工具从仓库进行安装，然后修改配置参数，这种方式简单又高效，一些公司也建立了自己的软件仓库，采用软件包管理的方式同样也存在一些问题，具体如下：

- ❑ 官方仓库安装的版本太老旧，跟不上新版。
- ❑ 安装方式的目录结构零散，不能完全按照自己的意愿组织文件。
- ❑ 自建仓库也比较难实现多版本、多配置的服务。
- ❑ 在自动化和运维可控性上较弱。
- ❑ 不方便定制各种优化参数。

在服务器规模较大的环境中进行逐台手工编译或安装的方式会造成很多问题：耗时费力，严重拖延业务的上线或变更时间，结果很难预期，很难保证每台都完全一样，会有很多意外情况。运维部署进程还会碰到一些需要解决的问题，比如：发布生产环境要控制安全，要做 DO 分离，不是每个人都可以去部署；软件要不断进行迭代升级和更新；配置文件要一致，不能跟进程部署完全耦合，也不能完全隔离分开；跟公司的 CMDB 等运维系统打通。这些用手工的方式都没办法做到。所以很多大型互联网公司都会选择自建包部署系统，接下来我们介绍一下 YY 自建包部署系统的设计思路。

### 3. 自建包部署系统

我所在的公司 YY 在服务端使用的是自建的包部署系统。大规模集群的部署必须要通过自动化来实现，只有通过自动化部署的平台，才能实现线上软件一致、版本一致、配置一



致、目录一致，这些都对后续的运维效率影响重大，至关重要，只有这样也才能实现与其他系统打通，与整个运维体系整合在一起。接下来简单介绍下建设思路。

我们定义的软件包指的是一组目标执行代码文件、脚本和配置文件，按照打包规范组成可以自行安装和自行部署的组合，打包过程如图 7-8 所示。

### 1) 运维目标

通过包部署系统来实现我们的运维目标。

- 统一管理：不用大家都去编译一个版本，一次编译，全网部署，全网一致。
- 发布效率：包部署的效率比每次编译和仓库安装快 N 倍，而且不容易出错，包括回滚的能力和效率。
- 管理配置：配置文件的修改也是经常会发生的事情，可以通过系统统一修改下发配置文件。
- 灵活的运维需求：可以定制前后置脚本，比如监控、定期任务，跟其他运维系统打通等。
- 有记录可供回溯：每一次的发布都有记录，方便管理、控制安全、后续审计等。

### 2) 包部署规范

打包和部署的工作不只是一件将软件文件组成一个包的事情，还包括一系列的规范。

- 打包规范：是指包括哪些内容、目录、版本、配置规范路径，文件名称，通用变量等。
- 部署规范：如何部署到目标服务器、路径、文件，对其他系统的依赖，启停脚本规范。
- 监控规范：需要哪些脚本，脚本名称规范、定期执行规范、上报规范、自动监控、拉起。
- 环境规范：OS 发行版本、系统库版本。

按规范打完包后接下来讲解如何部署软件包。

### 3) PHP 打包规范

具体到 PHP 包，我们的打包流程及其规范具体如下。

- 专人编译 PHP，全公司统一版本，编译参数是经过运维团队仔细考虑和优化过的，充分考虑公司的 Linux 发行版，不同 OS 发行版采用不同的包，收集了大部分 PHP 团队的意见，兼顾了大家的需求。
- 配置文件 `php.ini` 和 `php-fpm.conf` 均经过仔细优化，以适应公司的服务器机型，如 Web 类服务器的内存和 CPU 情况。
- 日志路径、配置路径、脚本路径、监听端口都统一规定好。
- 把核心扩展（MySQL、APC 等）编译进 PHP 可执行文件，常用的扩展（Redis、MongoDB、Memcached 等）编译成独立的 `.so` 文件，放在扩展目录下。
- 把所依赖的一些系统库文件，也放置到一个依赖目录，比如 `libjpeg`、`libevent` 等，总之把包发下去不会影响现在的系统库，也不会依赖太多的系统和第三方库。
- 写好了统一的监控脚本，在固定目录中定期执行监控脚本，并上报到公司监控系统。

### 4) PHP 包的升级和发布

首次打好的 PHP-FPM 包要能够满足当前的需要，后期可能会根据需要做一些调整。这种调整可以通过升级包版本的方式来完成，而不是重新打另外一个包。对于升级的包我们会分配一个新的包版本号。升级包可以替换包的任何部分。这样各个版本的包还是可以在线继续运行的，在包部署系统中也能进行启动、停止、升级等各种操作。

当业务需要往一台服务器中部署 PHP-FPM 包时，我们一般的流程是先到包部署系统中选择你的 PHP-FPM 需要的包和版本，选择或输入目标服务器 IP，点击部署就可以开始了。部署过程如图 7-9 所示：

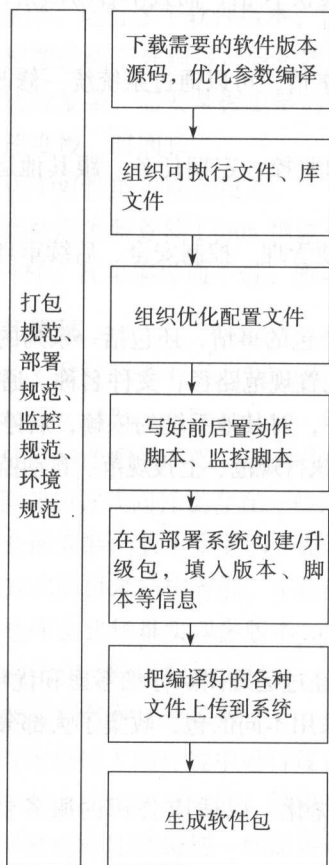


图 7-8 打包过程示意图

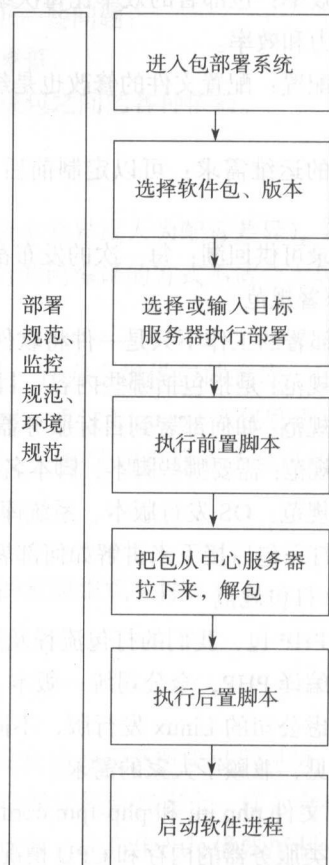


图 7-9 包部署过程示意图

不是每个公司都有能力开发这样的部署系统，了解了核心思路后通过一些简化的做法也能基本实现整个流程。打包过程还是一样，最后不用传到系统，而是以 tar 包的形式保存在集中服务器，部署的时候通过 Puppet、SaltStack、Ansible 等批量运维工具分发到目标服务器上，解压，执行前置，部署，后置脚本，部署监控脚本等。

说到大规模部署，目前容器技术也很热门，也有公司把 PHP 等进程直接做成 VM 镜像，

或者打成 Docker 镜像。通过 VM 和 Docker 的部署来实现进程的部署。容器技术属于另外的范畴, 这里不会展开来讲, 有兴趣的读者可以自行查找资料。

### 7.3.2 PHP 配置文件变更

配置文件是服务器软件的重要组成部分, 使用不同参数的配置文件可以满足各种特定的业务场景, 比如不同的服务器资源, 负载不同的业务。配置文件也是线上修改比较频繁的部分, 很多时候不需要变更软件, 而只需要修改配置文件。接下来我们讲解 PHP 配置文件的变更。

进行手工部署的时候, 也需要登录服务器修改配置文件, 当服务器有 10 台以上时, 不可能逐台去登录修改了, 必须通过 SaltStack、Ansible 等批量运维工具来实现。再进一步就要考虑建立配置文件管理系统。配置文件是运维标准化、自动化中非常重要的一环, 决定了软件的运行方式和运行状态, 修改配置文件是运维的日常工作。是否能够高效、准确又灵活地管理变更这些配置文件, 变更后不会造成混乱, 这是运维团队及运维平台是否成熟的体现。

PHP 有两个重要的配置文件, `php.ini` 和 `php-fpm.conf`。

`php-fpm.conf` 文件: 用于控制 PHP-FPM 管理进程的相关参数。比如工作子进程的数量、文件描述符数量、队列、监听端口等。

`php.ini` 文件: 用于控制 PHP 工作进程处理请求时的相关参数。配置项对应于工作进程, 对通过这个进程处理的每一个请求都有效。比如 PHP 工作进程启动时要载入哪些扩展、将 session 保存在哪里、此次请求最大的内存大小、错误日志、语言特性、可以上传多大的文件、日志文件的路径等。

那我们应该怎样快速修改这些配置文件呢, 有如下几种常见的做法。

□ 包部署系统管理配置文件: 将配置文件的变更当做一次软件升级。

□ 配置文件跟 PHP 代码一起发布: 发布业务代码时将配置文件也随之发布。

□ 独立的配置文件管理系统: 通过自动化运维系统变更配置文件, 首次部署软件时初始化了配置文件, 后续的变更都通过配置文件管理系统来进行。

不管是采用哪种配置文件管理方式, 都应该包含以下的核心功能。

(1) 配置查看: 查看配置文件的内容、配置变更历史、查看某版本的配置文件。查看的内容是保存在数据库中的。可以根据服务器和应用模块来查看。

(2) 变更部署: 一般是在默认配置文件的基础上修改参数, 修改了配置后可以快速下发到目标服务器, 并重启 PHP 进程。这里的目标服务器一般以应用模块为单位, 比如说登录模块有 4 台服务器, 一般会同时修改这个模块的全部服务器。当然有灰度能力就更方便了, 先发 1-2 台服务器, 成功了再发全部。

(3) 如果是与包部署系统结合的方式, 可以为每个包先做多个满足不同场景的配置文件。比如新的高性能服务器需要使用更多的进程数, 为它单独分配一个配置文件, 又比如为

某特殊的应用使用一份特殊的配置文件。测试环境、预发布环境、生产环境可以使用各自不同的配置文件。在部署包的时候选择需要的配置文件。

(4) 历史记录可以记录每次的变更历史、发布历史, 以及修改了的配置文件, 需要进行单独的管理。比如说某些服务器使用了特殊配置, 应该既能从应用模块或服务器查配置, 也能从配置查服务器和模块。

### 7.3.3 PHP 配置项

PHP 配置文件 `php.ini` 和 `php-fpm.conf` 中的配置项非常多, 因为本节的重点不在于此, 加上这类资料也比较容易获取, 因此我们仅用思维导图把这些配置项进行分类, 希望能帮助大家了解配置项 (`php-fpm.conf` 配置项的分类如图 7-10 所示, `php.ini` 配置项的分类如图 7-11 所示):

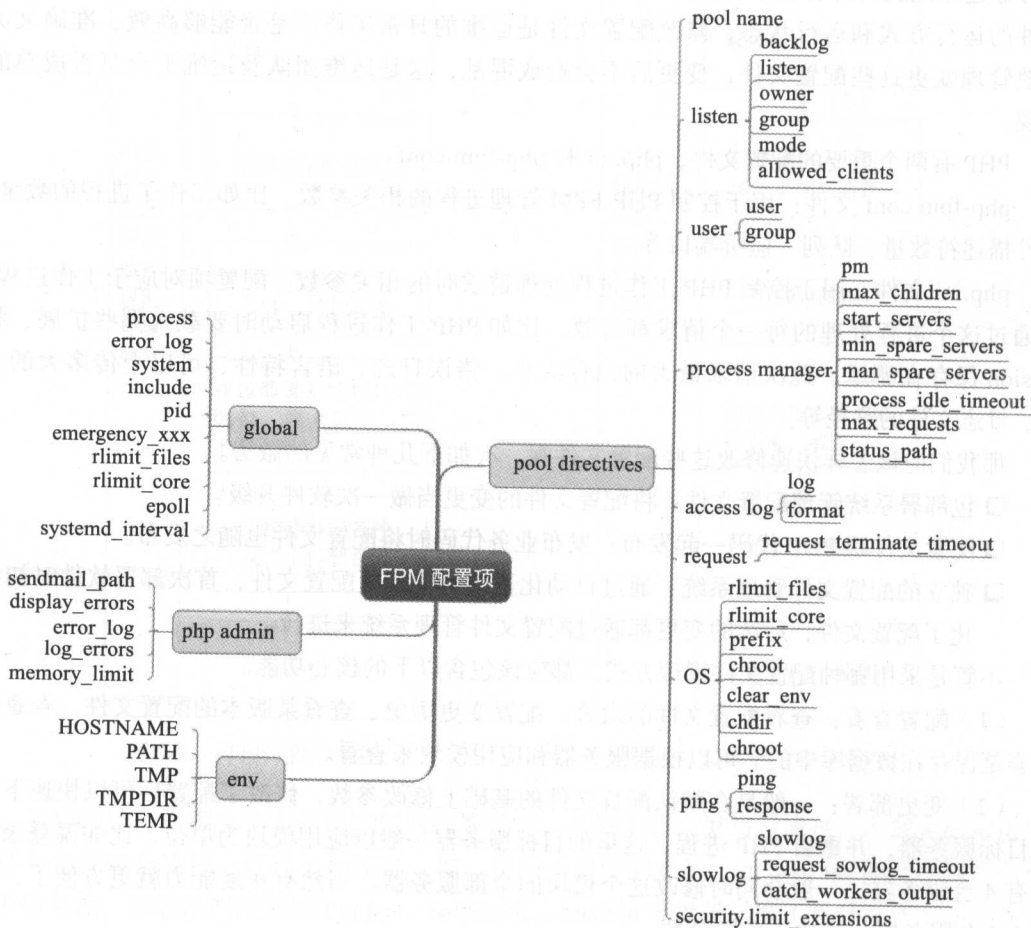


图 7-10 `php-fpm.conf` 配置项分类

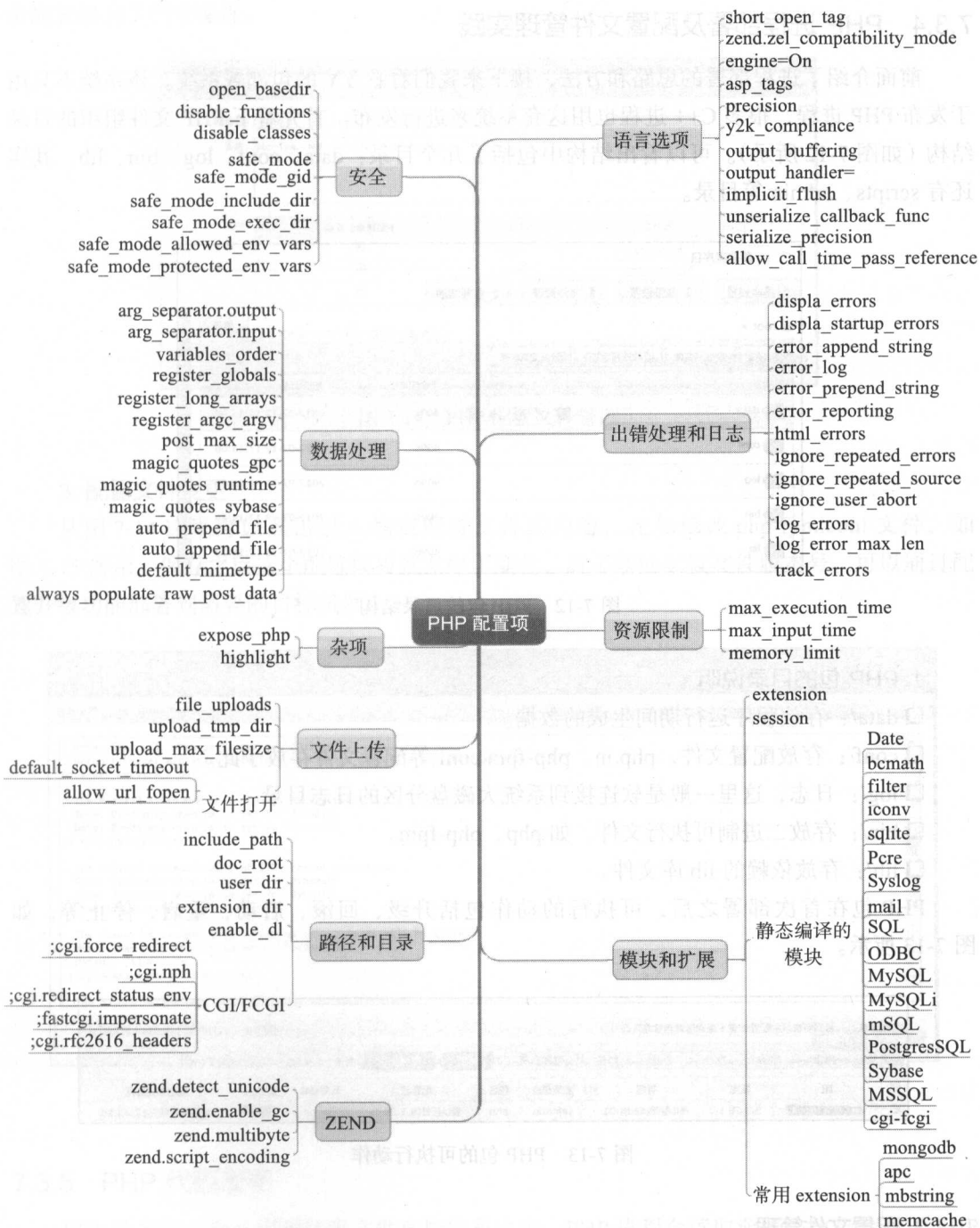


图 7-11 php.ini 配置项分类

7.3.4 PHP 进程部署及配置文件管理实践

前面介绍了进程部署的思路和方法，接下来我们看看 YY 的包部署系统。该系统不只用于发布 PHP 进程，很多 C++ 进程也用这套系统来进行发布。首先看下 PHP 文件组织的目录结构（如图 7-12 所示）。可以看出结构中包括了几个目录：data、conf、log、bin、lib，其实还有 scripts、admin 等目录。



图 7-12 PHP 包的目录结构

1. PHP 包的目录说明

- ❑ data/：存放程序运行期间生成的数据。
- ❑ conf/：存放配置文件，php.in、php-fpm.conf 等配置文件存放于此。
- ❑ log/：日志，这里一般是软连接到系统大磁盘分区的日志目录。
- ❑ bin/：存放二进制可执行文件，如 php、php-fpm。
- ❑ lib/：存放依赖的 lib 库文件。

PHP 包在首次部署之后，可执行的动作包括升级、回滚、启动、重启、停止等，如图 7-13 所示。

实例列表

返回当前版本配置管理 | 返回当前包管理页面

停止

运行中

操作全部

记录数(1)

升级

回滚

启动

重启

停止

卸载

配置变更

导出

仅导出IP

提示：点击字段名可以实现当前页内排序

	IP	版本	机房	业务模块	包名	配置名	最后操作	操作人	最近操作时间
<input checked="" type="checkbox"/>	192.168.1.1	5.3.28.1	佛山智慧城多线-01	tafnode	php	默认配置(5.3.28.1)	install	admin	2012-07-12 17:43:26

图 7-13 PHP 包的可执行动作

2. 配置文件管理

YY 包部署系统中配置了文件模板编辑功能，可为每个场景或应用生成一份定制的配置文件。如图 7-14 所示，这里生成了一份名为“php-fpm.conf for highperf”的 PHP 配置，这



份配置包含了三个文件。

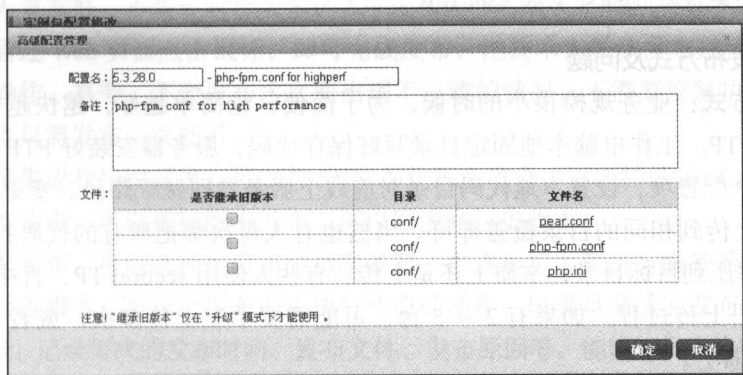


图 7-14 PHP 配置文件管理

### 3. 配置文件变更

从图 7-14 这里可以点击进入编辑配置文件的内容，比如修改 php-fpm.conf 文件，如图 7-15 所示。保存之后这个配置模板就发生了变化，有了新的配置文件版本号，可以通过配置升级功能部署到需要的目标机器上。

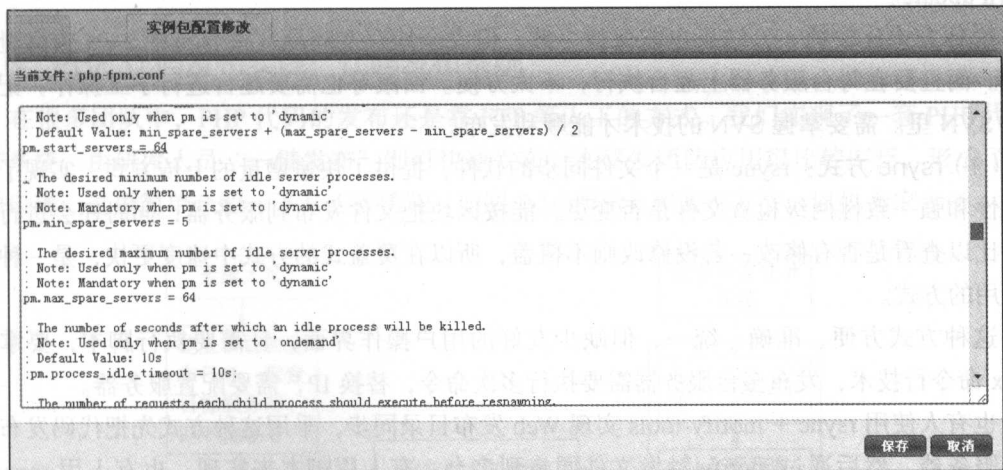


图 7-15 配置文件变更

## 7.3.5 PHP 代码发布

PHP 的发布一般是把源代码文件直接发布出去，PHP 进程会读取源码并执行，所以 PHP 发布起来也是最简单的，简单到通过 FTP 软件把 PHP 文件上传到服务器上就可以了。如果只有一两台服务器，通过 FTP 传上去也许可行，但是当服务器很多的时候，就不能靠手工



FTP 了。从专业的运维角度来看，仅把文件上传到服务器上是不够的，而且还可能会出现各种问题。我们先来看看几种常见的发布方式。

### 1. 常见的发布方式及问题

(1) **FTP 方式**：业务规模很小的时候，为了简便，越简单越好，越快越好，最常见的发布方式就是 FTP。工作电脑本地固定目录写好保存代码，服务器安装好 FTP 服务，在 FTP 软件中建好 FTP 配置项，设置本地代码目录对应线上服务器网站根路径。变更了的代码只要把文件或目录上传到相同的目录覆盖即可。当然也有人每次都把所有的代码打成 tar 包，上传到服务器再解压到目标目录，本质上还是上传。有些人使用 ssecureFTP，直接在 secureCRT 软件中进入代码上传过程。如果有 2 ~ 3 台，可能需要手工上传多次，或者从一台服务器 scp 到别的服务器上。

这种方式存在一些问题：不知道谁在什么时候会发布，多人合作容易互相覆盖；没有代码版本管理，若发布出现故障可能会找不到之前正确的版本；也没有发版的概念，随意性太强，无法快速回滚；没有记录，找不到发布历史；对一两台服务器一般没有问题，但多几台服务器就会无法接受。

(2) **svn update 方式**：有些团队有使用 SVN 等版本管理工具，使用 SVN 进行管理，于是发布也随之改变。有些团队把代码提交到 SVN 服务器，在所有服务器的文件根目录中执行 svn update。

这种方式比较统一，结果也能保持一致，但安全性存在问题，很容易把 SVN 信息泄露出去。而且要在每台服务器上逐台执行，不太方便。回滚等也需要逐台进行手工操作，记录只在 SVN 里。需要掌握 SVN 的技术才能顺利发布。

(3) **rsync 方式**：rsync 是一个文件同步的软件，提供了快速增量的上传方式，实现了弱一致性和强一致性两级检查文件是否变更，能按区块把文件发布到服务器，同时还会进行文件对比以查看是否有修改，若没修改则不覆盖，所以在覆盖式的方式中速度更快，是一种普遍使用的方式。

这种方式方便、准确、统一，但缺少友好的用户操作界面，且需要执行脚本，要掌握 Linux 命令行技术，发布多台服务器需要执行多次命令，替换 IP，需要配置服务器。

也有人使用 rsync + inotify-tools 实现 Web 发布目录同步，采用这种方式先把代码发布到一台服务器，然后通过 inotify 触发文件同步到多台。有人用脚本来实现，也有人用 sersync 等类似软件来实现。

还有人把 rsync 与 SVN 结合起来，发布时将 SVN 更新到一个目录，然后 rsync 把这个目录同步到多台服务器上。

### 2. 代码发布系统

手工部署无论如何都会存在较多问题，即使有完善的文档，也很难完全避免发生意外。建设发布系统才是发布代码的正确姿势，为了解决发布的众多问题，一个优秀的发布系统需

要具备以下特性。

- ❑ 对发布人员友好：不需要掌握太多的 Linux Shell 技术，不需要记住 rsync 复杂的参数配置。通过 Web 页面点击鼠标，即可完成发布，把技术人员从发布过程中解放出来。
- ❑ 可重复操作，幂等：不会发布多次而出现不一致的情况，不需要反复执行命令，发布到多台也只需发布一次即可。
- ❑ 支持持续集成和持续部署：可以与 SVN 等代码库整合，可以持续集成，持续部署。
- ❑ 支持版本管理：可以选择版本发布、可以回滚到历史版本。
- ❑ 支持灰度发布：可以选择先发布少量服务器，再逐步扩大到所有服务器。
- ❑ 支持前后置脚本：可以在发布前后执行相应的动作，以满足灵活的发布需求。
- ❑ 发布记录：记录每次的发布时间、发布文件、发布原因等，能统计发布次数、发布耗时。
- ❑ 支持多个环境发布：同一套代码可以发布到开发环境、测试环境、生产环境，可为不同的环境配置不同的发布任务。

发布系统自动化是运维自动化的重要组成环节，如果能把自动化发布做好，那么很多运维工作才好开展，也能更好地推动下一步自动化的工作。有必要将自动化发布在团队中推行开来，自动化运维能带来很多收益。自动化发布的目标是与高度自动化的测试、自动化部署和全面的配置管理结合在一起，能够实现一键部署，就可以把代码部署到开发、测试和生产环境中，且能在出现故障时快速回滚，加上各种管理功能。

### 7.3.6 PHP 代码发布实践：代码发布系统

在很多团队中，PHP 代码的发布还是靠 FTP 等人工的方式。我们实现了一套 PHP 代码发布系统，由研发人员“一键发布”即可快速发布，在研发团队应用得比较广泛，形成了较好的口碑。这里简单介绍一下其工作流程（如图 7-16 所示）和实现方法，以供大家参考。

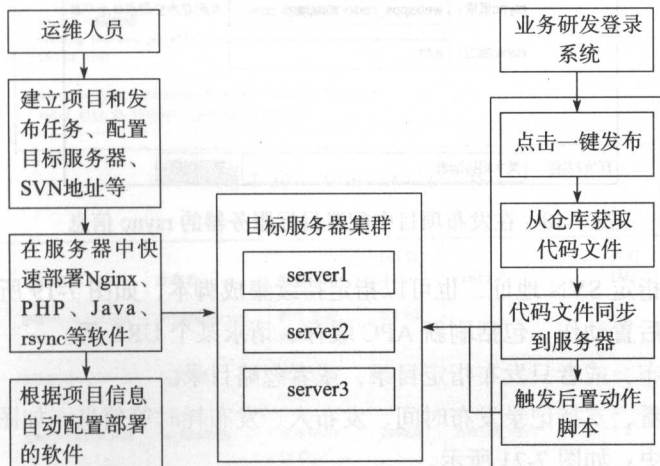


图 7-16 发布系统的流程图

这是一个简单的系统，主要功能包括如下几点。

- 可为不同的环境创建不同的发布任务，如测试、开发、生产、灰度环境等，如图 7-17 所示。

任务列表				
序号	任务名称	执行的操作	最后发布	操作
1	lol内战分组 访问站点	发布到lol内战分组 [全部发布] svn更新	svn版本号: 187852 dw_c[redacted]zao 2015-12-01 09:16:46	一键发布 日志 修改 预发布
2	美术组排期表 访问站点	发布到美术组排期表 [全部发布] svn更新	svn版本号: 187851 dw_c[redacted]ao 2015-12-01 09:15:15	一键发布 日志 修改 预发布
3	技术组排期表 访问站点	发布到技术组排期表 [全部发布] svn更新	svn版本号: 187851 dw_c[redacted]ao 2015-12-01 09:14:39	一键发布 日志 修改 预发布

图 7-17 发布系统 - 任务列表

- 指定目标服务器：rsync 模块路径、用户名、密码端口等信息，如图 7-18 所示。

环境1名称

技术组排期表

收起配置

环境1配置

rsync IP: [redacted]128.5 [redacted]128.6 多个ip换行隔开

rsync帐号: webapps 若配置为空则忽略此环境

rsync密码: [redacted]kP13VDRRND 若配置为空则忽略此环境

rsync模块: webapps\_code/[redacted].com 若配置为空则忽略此环境

rsync端口: 873

环境2名称

美术组排期表

展开配置

图 7-18 在发布项目中配置目标服务器的 rsync 信息

- 创建项目，指定 SVN 地址，也可以指定持续集成脚本，如图 7-19 所示。
- 指定前置、后置动作，包括刷新 APC 缓存、请求某个 URL 等。
- 指定全部发布，或者只发布指定目录，或者忽略目录。
- 发布过程查看，支持记录发布时间、发布人、发布耗时等信息，如图 7-20 所示。
- 查看发布历史：如图 7-21 所示。
- 支持指定用户权限，跟公司账户体系打通。

项目名称

任务名称

发布到测试环境

任务权限

请输入拥有权限名单

项目app\_id

项目app\_id

用于调试和性能分析

使用SVN

☒ 是 ☐ 否

SVN根地址

SVN用户名

请输入SVN用户名

使用

com的项目无需填写

SVN密码

不修改无需填写

使用

com的项目无需填写

行前重命令

☒ 是 ☐ 否

发布目录

/

目录或文件列表

protected/data

子目录也会设置为读写权限，目录前后部不需要斜杠"/"，回车换行分隔

同步方式

☒ 全部同步 ☐ 指定忽略 ☐ 指定文件或目录

代码发布到

☒ 测试环境

231

刷新CDN

☐ 是 ☒ 否

图 7-19 创建发布任务详情

100% 耗时: 1.2 秒

从SVN导出代码

svn: 'http://svn.duowan.com:9999/svn/'  
没有文件被更新  
Checked out revision 206105.

正式环境IP 183.6 RSYNC同步代码

IP:183.6  
sending incremental file list  
xhprof\_html/  
  
sent 727 bytes received 19 bytes 1492.00 bytes/sec  
total size is 1374561 speedup is 1842.58

图 7-20 发布过程及结果

序号	任务名称	发布者	发布时间	svn版本号	svn日志	耗时(秒)	结果
1	static正式环境	dw	2016-06-27 19:15:37	206103		4.6	成功
2	static正式环境	dw	2016-06-27 16:22:27	206052		4.3	成功
3	static正式环境	dw	2016-06-27 16:19:32	206050	消息通知改造	5.8	成功

图 7-21 发布历史

## 7.4 PHP 性能分析

### 7.4.1 性能问题概述

随着网站用户的持续增长，如何以有限的资源尽可能地服务于更多的用户呢，PHP 性能优化在这里就显得特别重要。PHP 具有入门简单、容错性强、性能较高的特点，也很容易由于测试不完备、代码随意、对性能不够重视等原因导致上线后出现各种问题，且不容易在第一时间就能找到性能瓶颈点，没有经验的人甚至无从下手去收集全面的信息加以分析。由于 PHP 运维人员面对的是生产环境，因此掌握 PHP 的性能分析和解决性能问题是 PHP 运维人员必备的非常重要的能力。

影响 PHP 性能的点有很多，我们先来了解“性能”这一概念。性能的本质是延时和吞吐率，即所谓的 latency 和 throughput。吞吐率是指在单位时间里完成的工作量，延迟是指一个请求从开始到完成所用的时间，有时也会认为与响应时间是同一概念。

❑ 延时 / 响应时间：一个 PHP 的请求 (request) 需要花费多长时间来处理，PHP 执行需要多长时间。影响响应时间的因素包括 PHP 代码的执行时间、CPU、内存、磁盘 IO、网络 IO、SQL 请求等。

❑ 吞吐率 (对 PHP-FPM 来说是 RPM : Request Per Minute)：吞吐率跟响应时间成反比，响应时间越短，单位时间内可处理的请求次数越多，即吞吐率越大。

❑ 错误率 / 失败率：PHP 处理缓慢会导致超时或触发错误、异常，性能问题也体现在请求的错误率升高。

延迟和吞吐可以体现出 PHP 应用的性能，我们可以将它们作为运维质量的指标来进行统计分析。从业务请求来看，我们要判断一个请求的响应时间少于多少是合理的 (比如 1.5s)，超过阈值则可用性指标会下降，吞吐率下降到多少是异常的，要触发预警和告警，反馈给业务研发。错误是直接影响用户使用的错误结果，用户看到的可能是 502、500、503 等错误。错误率上升了一定是出现了故障。

造成性能问题的原因多种多样，深入了解 PHP 的性能问题是我们快速定位问题点、快速解决问题的基础。接下来我们分析下 PHP 的性能问题。

如何才能知道网站和 PHP 能承载多大的访问量呢？正常的情况下延时是多少呢。我们通常用 ab (Apache Bench) 工具来分析，ab 可以模拟用户对网站发起请求，常用参数为 -n、-c，参数 -n 表示总请求次数，-c 表示同时发起的并发请求数。例如要测试某网站的性能，命令如下：

```
ab -n 300 -c 20 http://www.test.com/index.html
```

关注结果中的 “Requests per second” 和 “Time per request”：

```
Requests per second:      285.54 [# /sec] (mean)
Time per request:         1053.502 [ms] (mean)
```

可以发现平均响应时间大概在 1s 左右，并发数在 285 左右。结果中还有很多有用的信息，有兴趣的读者可以查阅相关资料。

## 7.4.2 PHP 性能问题

广义的 PHP 性能是指 PHP 站点的整体性能（如图 7-22 所示），是指整个站点软硬件架构，包括硬件、操作系统、Web 服务器、PHP、到后端的缓存、存储及调用的外部接口性能等。狭义的 PHP 性能是单指 PHP 自身（包括 PHP 进程和 PHP 脚本）的性能（图 7-22 中的灰色背景部分）。

网站整体输出的性能也就是整个网站架构的性能，从外部来看，URL 每秒的请求数和响应时间就是站点的整体性能。7.1 节中讲整个请求流程时讲到请求要经过很多环节，每个环节都可以用吞吐和延迟两个指标来衡量性能。例如我们常说的 Nginx 能支持每秒多少个请求，MySQL 支持多少个并发查询，磁盘支持多少 IOPS 等。分布式架构正是为了解决某个环节的性能瓶颈问题而设计的，比如 Nginx 集群解决 Web 层性能问题，PHP 集群解决应用层性能问题，MySQL 主从、分库分表等措施都是为了通过集群来解决数据存储层的性能问题，分布式分散压力都是使得本层的性能不会成为整个站点的瓶颈。本节的重点是讲解其中的应用服务这一层，也就是 PHP-FPM 这一层的性能问题。

一个 PHP 网站可能包括多个服务（如登录、首页、列表、上传服务等），也有给其他服务调用的 API。每个服务又由一系列的类或函数调用所组成，函数要访问外部接口、外部存储、本地 IO，也包括 PHP 本身执行过程中内存和 CPU 的消耗。每一个函数都可能是性能瓶颈点，也可能在某些条件下引发性能问题，从而导致整个服务，甚至整个业务出现故障。图 7-23 展示了逐级细化后的 PHP 性能组成部分。

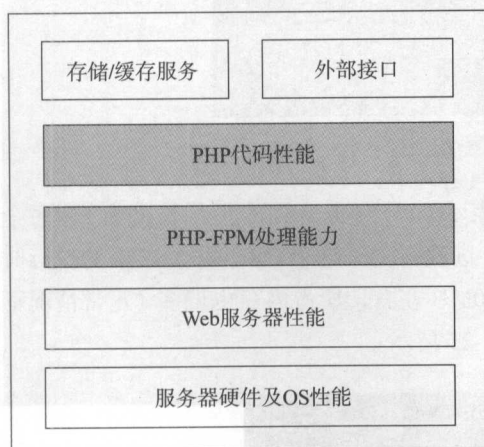


图 7-22 PHP 广义和狭义性能

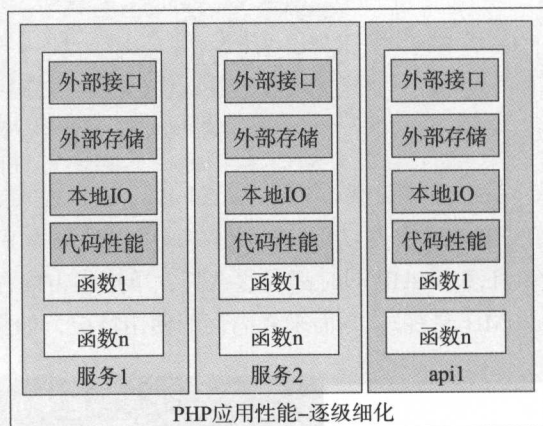


图 7-23 PHP 应用性能 - 逐级细化

我们可以以不同的视角来看待整个网站和 PHP 的性能问题，从宏观定位出问题的是



哪一层级和哪一个服务，然后再到微观层面看是哪个服务的哪个函数或接口，消耗了什么资源。

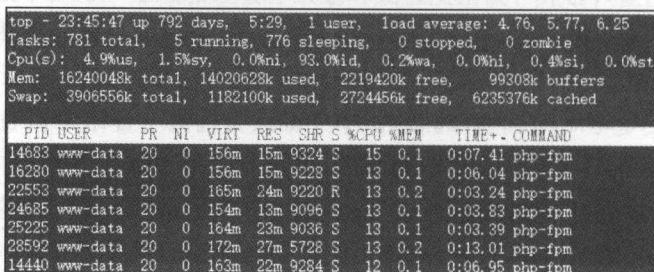
### 7.4.3 性能分析方法

性能问题牵涉到的技术点很多，且发生在生产环境中，没办法离线进行慢慢检查，再加上与系统中的其他软件互相依赖互相影响，所以定位起来不太容易。当 PHP 发生了性能问题，一定是有某个地方达到了瓶颈，我们要做的就是找到这个点，然后消除它。本节将讲解通用的 PHP 性能分析方法。首先我们分几个层级来进行分析，先从操作系统的层级来看看系统的负载，再看看 PHP 自身输出的运行状态，比如各种日志，这些日志是 PHP 运行情况的表现，然后通过 PHP 内部的运行细节，发现 PHP 运行过程中的性能瓶颈。

#### 1. 系统 / 进程级分析

硬件和操作系统是 PHP 进程正常运行的基础，如果系统本身资源不够或其他进程抢占了太多的资源，那么肯定会导致 PHP 发生性能问题，所以本节先讲解系统负载和 PHP 进程资源占用的分析方法。

(1) 系统负载分析：top 命令是最基本的 Linux 命令，可通过 top 命令查看机器整体负载和 PHP 进程的资源消耗情况，如图 7-24 所示：

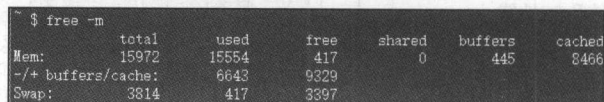


```
top - 23:45:47 up 792 days, 5:29, 1 user, load average: 4.76, 5.77, 6.25
Tasks: 781 total, 5 running, 776 sleeping, 0 stopped, 0 zombie
Cpu(s): 4.9%us, 1.5%sy, 0.0%ni, 93.0%id, 0.2%wa, 0.0%hi, 0.4%si, 0.0%st
Mem: 16240048k total, 14020628k used, 2219420k free, 99308k buffers
Swap: 3906556k total, 1182100k used, 2724456k free, 6235376k cached
```

PID	USER	PR	NI	VIRT	RES	SHR	S	%CPU	%MEM	TIME+	COMMAND
14683	www-data	20	0	156m	15m	9324	S	15	0.1	0:07.41	php-fpm
16280	www-data	20	0	156m	15m	9228	S	13	0.1	0:06.04	php-fpm
22553	www-data	20	0	165m	24m	9220	R	13	0.2	0:03.24	php-fpm
24685	www-data	20	0	154m	13m	9096	S	13	0.1	0:03.83	php-fpm
25225	www-data	20	0	164m	23m	9036	S	13	0.1	0:03.39	php-fpm
28592	www-data	20	0	172m	27m	5728	S	13	0.2	0:13.01	php-fpm
14440	www-data	20	0	163m	22m	9284	S	12	0.1	0:06.95	php-fpm

图 7-24 top 命令查看系统及进程负载

系统整体负载（5 分钟 load avg）在 5 左右，表示 CPU 任务队列的平均长度为 5，每个 PHP-FPM 进程占 CPU 13% 左右，可以理解为 PHP 进程占用了这个 CPU 线程 13% 的执行时间，且多个 PHP 同时都占这么高；单进程占内存 30MB 左右，所占内存也偏多（正常情况下是 20MB 左右）。下面来看内存的使用情况，如图 7-25 所示：



```
$ free -m
```

	total	used	free	shared	buffers	cached
Mem:	15972	15554	417	0	445	8466
-/+ buffers/cache:		6643	9329			
Swap:	3814	417	3397			

图 7-25 服务器内存使用情况



从图 7-25 所示的内存使用情况来看,本机有 16GB 的内存,已基本被用完,但有 8466MB 是被用在了缓存上,实际只使用了 6643MB,所以内存实际上还是比较充裕的。下面来看服务器 IO 使用情况,如图 7-26 所示:

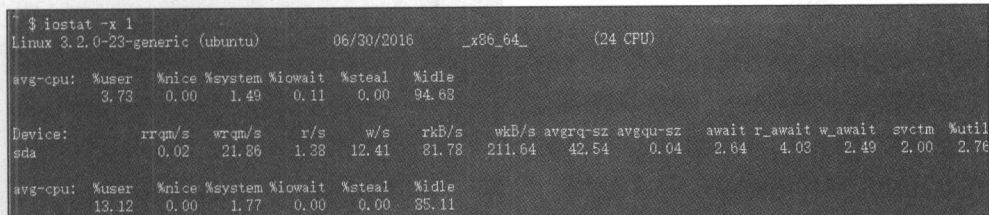


图 7-26 服务器 IO 使用情况

从图 7-26 分析的服务器 IO 使用情况可知，IO 利用率在 2.76%，也是很低的。还有其他的很多信息，限于篇幅不能逐一展开。总之这台服务器目前负载较低，有一定的业务访问，系统负载问题并不突出。

(2) **系统调用分析:** 所有软件对磁盘、内存、网络等硬件的操作都是通过操作系统来执行的, 而系统调用是操作系统内核提供给用户态进程的接口方法。strace 工具可以帮助查看用户进程调用的那些系统调用 (如图 7-27), 比如磁盘读写、网络连接、网络读写、内存分配等操作及其耗时, 从而进一步分析进程的资源消耗, 进而有针对性地改进代码, 优化程序。

命令格式: `strace -p <PID> -T`

其中 PID 是 PHP-FPM 工作进程的进程 ID。

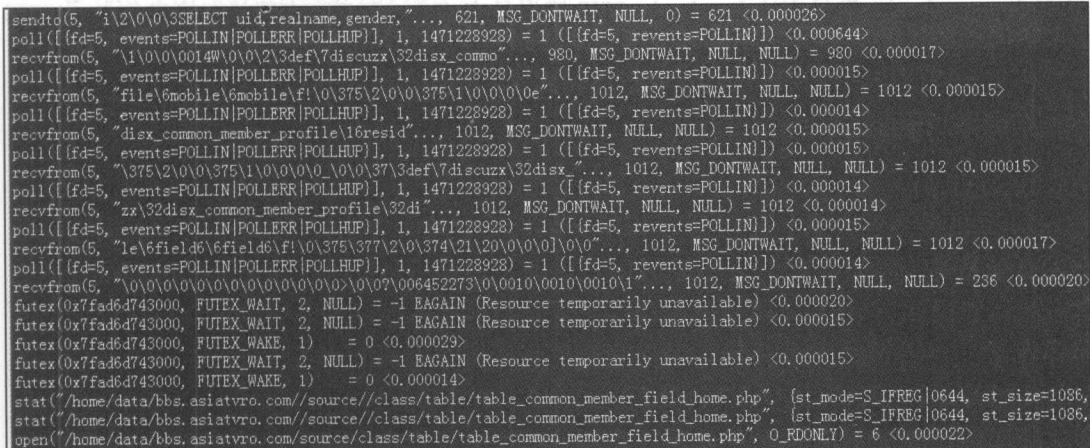


图 7-27 通过 strace 观察进程的系统调用

从图 7-27 可以看出大部分系统调用都在执行读取文件和从 MySQL 获取数据，且 MySQL 返回的字节数都有 1000 多字节，说明 SQL 返回的结果较大，但每次调用耗时还是

比较短的 (0.000015s), 如果这里出现较长时间, 很可能就是瓶颈所在。

strace 还能统计一段时间内各个系统调用的执行次数和消耗时间, 命令如下, 结果如图 7-28 所示:

```
$strace -p <PID> -c
```

从图 7-28 可以看出 stat (获得文件的状态) 占了 27% 的时间, poll (监测网络连接队列的事件) 占了 27%。说明主要负载还是在查找文件和网络连接处理这两方面。解决办法就是和研发一起找到读取文件最多的代码部分, 通过优化 PHP 代码进行解决, 如果上面读取文件太多, 那么从上面

strace 的输出可以粗略看出是读取的 PHP 文件太多, 应该能分析到是每次都读取了 PHP 文件, 而且遍历了多层路径, 还指明了具体的文件和路径。所以加入 PHP 的 Opcode 缓存, 减少对 PHP 代码文件的读写也许是最好的解决办法。另外在 PHP 代码中 include 其他 PHP 文件时, 用了相对路径也会由于路径遍历造成多次读取文件, 将 include 文件改为绝对路径是更好的方式。

% time	seconds	usecs/call	calls	errors	syscall
27.63	0.000121	0	1283		stat
27.40	0.000120	0	650		poll
15.53	0.000068	0	1017		mmap
12.33	0.000054	0	142		sendto
9.36	0.000041	0	3051		fstat
3.88	0.000017	0	1039		close
3.88	0.000017	0	653		recvfrom
0.00	0.000000	0	50		read
0.00	0.000000	0	33		write
0.00	0.000000	0	1017		open
0.00	0.000000	0	123		lstat
0.00	0.000000	0	1017		mmap
0.00	0.000000	0	30		rt_sigaction
0.00	0.000000	0	10		rt_sigprocmask
0.00	0.000000	0	17		1 access

图 7-28 strace 统计系统调用的分布情况

## 2. PHP 运行状态

PHP 充分考虑了运维的友好性, 可以把运行过程的内部状态通过日志和接口反馈出来, 让研发人员和运维人员得以了解 PHP 的运行状态。主要方式有 PHP-FPM 的错误日志、访问日志 (access log)、慢日志 (slow log)、PHP 错误日志 (error log)、内部状态接口 (php status) 等。接下来讲解如何根据各种日志和状态分析 PHP 性能问题。

### 1) 访问日志分析每个请求的耗时

如图 7-29 所示, access log 可以查看每次 PHP 的访问日志, 包含了请求方法, 请求的 PHP 文件及参数、返回码等, 也有 CPU、内存、时间消耗等。

```
"GET /forum.php?mod=forumdisplay&fid=10&page=2&mobile=yes" 200 24.480 2816 81.70%
"GET /forum.php?mod=forumdisplay&fid=2&page=4&mobile=yes" 200 23.802 2816 126.04%
"GET /forum.php?mod=viewthread&tid=1139683&mobile=yes" 200 74.736 3328 93.66%
"GET /forum.php?authorid=21269&mod=viewthread&tid=1112041" 200 31.853 3072 62.79%
"GET /forum.php?mod=viewthread&tid=959888&extra=page%3D1&page=2" 200 37.606 3328 79.77%
"GET /forum.php?mod=viewthread&tid=591715&extra=page%3D1&page=1" 200 34.407 3328 87.19%
"GET /forum.php?mod=forumdisplay&fid=10&filter=author&orderby=dateline&mobile=yes" 200
"GET /forum.php?mod=viewthread&tid=902893&extra=page%3D1&page=1" 200 70.767 5120 84.79%
"GET /forum.php?mod=forumdisplay&fid=10&mobile=yes" 200 22.721 2816 88.02%
"GET /forum.php" 200 8.133 1536 122.96%
```

图 7-29 PHP access log 性能分析

从图 7-29 可知所有请求都是成功状态 200, 大部分请求耗时都在 20ms 左右 (注意, 这个性能时间只包括 PHP 脚本本身的执行时间, 也就是 PHP 字节码执行时间的总和, 不包括 IO 时间和等待 SQL 的时间), 内存 3KB 左右, CPU 在 60 ~ 85% 之间。取最繁忙的 1 分钟

或一段时间内的一个平均值，可以得知每分钟请求次数（Request Per Minute, RPM），如图 7-30 所示：

```
root@ubuntu:~# grep " 26/Sep/2015:22:17" /data/weblog/php/web2.access.log |wc -l
49012
```

图 7-30 PHP access log 性能分析，计算 RPM

从图 7-30 可以看出一分钟处理了 49012 个请求，也就是每秒 800 次左右。

可以在 `php-fpm.conf` 文件中配置访问日志的格式，PHP 提供了丰富的选项来获得所需要的信息，具体如下。

- %%: 表示百分号 (%)。
- %C: 请求消耗的 CPU 百分比。
- %d: 请求消耗的时间。
- %e: 环境变量或服务器变量，与 `$_ENV`、`$_SERVER` 的效果一样。
- %f: 脚本文件名。
- %l: POST 请求的内容长度。
- %m: 请求的方法 GET/POST。
- %M: 这个 PHP 工作进程分配的内存的峰值。
- %n: 工作进程池名称，以区别多个进程池的日志。
- %o: 输出 HTTP HEADER 信息。
- %p: 工作进程的 PID。
- %P: PHP-FPM 管理进程 PID。
- %q: 查询字符串。
- %Q: 问号。
- %r: 请求的 URI 信息。
- %R: 客户端的 IP，一般会显示为 Nginx 的 IP。
- %s: 返回状态码。
- %t: 收到请求的服务器时间。
- %T: 写日志的时间，可以理解为请求完成的时间。
- %u: remote user, 远程用户。

系统默认是 `access.format = "%R - %u %t \"%m %r\" %s"`，也就是打印远程 IP、远程用户、请求时间、请求方法、请求 URI、返回状态码，不带性能数据。

图 7-29 中日志格式的配置如下：

```
access.format = "%R - %u [%{Y-m-d:H:M:S}t] %{REMOTE_ADDR}e \"%m %r%Q%q\"
%s %f %{seconds}ds %{megabytes}M %{user}C%{system}C%{total}C%"
```

PHP 访问日志非常有用，经常查看有利于持续发现某些性能比较差的服务和请求。

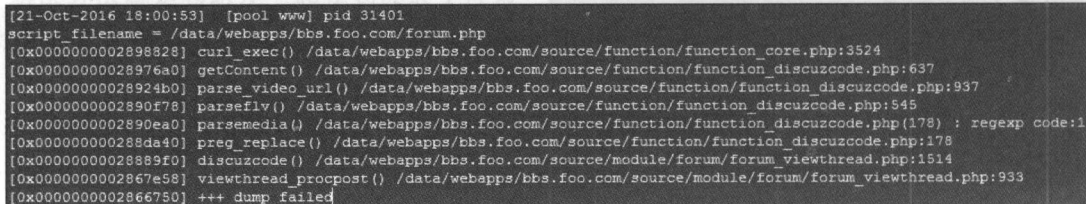
## 2) 慢日志分析性能

通过慢日志 (slow log) 可以看到那些超过规定时间的请求, 包括慢的代码文件及函数级的调用栈, 可以在 `php-fpm.conf` 中进行配置:

```
slowlog = /data/weblog/php/$pool.slow.log
request_slowlog_timeout = 3
```

指定慢日志的文件路径和请求出现在慢日志中的临界值, 若一个请求超过了这个时间 (3s) 即认为是慢请求。慢日志的超时时间可根据日常情况来定, 一般 2 ~ 3s 即可。

slowlog 信息展示如图 7-31 所示:



```
[21-Oct-2016 18:00:53] [pool www] pid 31401
script_filename = /data/webapps/bbs.foo.com/forum.php
[0x0000000002898828] curl_exec() /data/webapps/bbs.foo.com/source/function/function_core.php:3524
[0x00000000028976a0] getContent() /data/webapps/bbs.foo.com/source/function/function_discuzcode.php:637
[0x00000000028924b0] parse_video_url() /data/webapps/bbs.foo.com/source/function/function_discuzcode.php:937
[0x0000000002890f78] parseflv() /data/webapps/bbs.foo.com/source/function/function_discuzcode.php:545
[0x0000000002890ea0] parsemedia() /data/webapps/bbs.foo.com/source/function/function_discuzcode.php:178 : regex code:1
[0x000000000288da40] preg_replace() /data/webapps/bbs.foo.com/source/function/function_discuzcode.php:178
[0x00000000028889f0] discuzcode() /data/webapps/bbs.foo.com/source/module/forum/forum_viewthread.php:1514
[0x0000000002867e58] viewthread_procpst() /data/webapps/bbs.foo.com/source/module/forum/forum_viewthread.php:933
[0x0000000002866750] +++ dump failed
```

图 7-31 PHP-FPM slow log

图 7-31 中慢日志显示了时间、进程池、工作进程 pid、PHP 文件名。接下来是 PHP 请求的内部调用栈, 函数从下往上顺序调用。可以看到入口函数是 `viewthread_procpst`, 此函数调用栈是 `viewthread_procpst` → `discuzcode` → `preg_replace` → `parsemedia` → `parseflv` → `parse_video_url` → `getContent` → `curl_exec`, 发现 PHP 执行慢的根本原因是 `curl_exec` 函数, 找到 `function_core.php` 文件的第 3524 行, 分析 `curl_exec` 请求了哪个 URL 即可知道是哪个外部服务有问题。

## 3) PHP-FPM status 分析性能

PHP-FPM 比之前的 FastCGI 管理工具强大的地方也是因为它自带了一些内部的信息的工具, `php status` 能较好地显示工作进程的信息, 查看 PHP 内部运行过程中的状态。

在 Nginx 中配置 location:

```
location ~ ^/php_status$
{
    include fastcgi_params;
    fastcgi_pass 127.0.0.1:9000;
    fastcgi_param SCRIPT_FILENAME $fastcgi_script_name;
}
```

在 `php-fpm.conf` 中配置:

```
pm.status_path = /php_status
```

Nginx 收到 `/php_status` 请求, 转给 PHP-FPM 的服务, PHP 内部若发现匹配到配置项

pm.status\_path, 即返回状态信息。通过浏览器访问 Web 页面或通过 curl 访问状态路径 (status path) 时, 例如在浏览器中打开 [http://www.foo.com/php\\_status](http://www.foo.com/php_status), 或者 curl [http://www.foo.com/php\\_status](http://www.foo.com/php_status), 可看到如图 7-32 所示的信息:

```
[root@localhost ~]# curl http://127.0.0.1/status
pool:          www
process manager: dynamic
start time:    04/Sep/2016:03:13:12 +0800
start since:   4043475
accepted conn: 1994
listen queue:  0
max listen queue: 129
listen queue len: 0
idle processes: 12
active processes: 93
total processes: 105
max active processes: 128
max children reached: 0
slow requests: 71656
```

图 7-32 PHP-FPM status

下面来解释一下各个字段的意思。

- ❑ pool: PHP-FPM 工作进程池的名称, 默认为 www。
- ❑ process manager: 进程管理方式, 可取值为 static、dynamic 或 ondemand。
- ❑ start time: 启动日期, 如果重载了 PHP-FPM, 则时间会更新。
- ❑ start since: PHP-FPM 运行时长, 从启动到当前的时长。
- ❑ accepted conn: 当前进程池接受的请求数。
- ❑ listen queue: 请求等待队列, 如果这个值不为 0, 那么要增加 FPM 的进程数量。
- ❑ max listen queue: 请求等待队列最高的数量, 曾经达到的最大的请求等待队列的长度。
- ❑ listen queue len: socket 等待队列的长度。
- ❑ idle processes: 空闲进程数量。
- ❑ active processes: 活跃进程数量, 表示有多少工作进程在处理请求。
- ❑ total processes: 总进程数量。
- ❑ max active processes: 最大的活跃进程数量 (从启动 PHP 开始计算)。
- ❑ max children reached: 工作进程数量达到的最大数量, 如果这个数量不为 0, 那就说明你的最大进程数量太小了, 需要调大一点。
- ❑ slow requests: 慢请求的数量, 如果这个数字较大, 则有必须去分析性能问题了。

从图 7-32 中可以看出一些信息: 一共有 105 个进程, 93 个处于活跃状态, 最大的活跃进程数达到 128, 这个服务器的负载较高。PHP 进程启动有一个月了, 慢日志有 71656 条, 性能问题较严重。最大等待队列达到 129 个, 说明有些用户访问起来有点慢了。

status 获取到的是 PHP 进程的总体信息, PHP-FPM 同时还提供了每个工作进程的详细信息。在 URI 中加上 ?full 参数 ([http://www.foo.com/php\\_status?full](http://www.foo.com/php_status?full)), 可以指定返回的格式 (json/html/xml), 默认是普通文本格式:

```
http://www.foo.com/php\_status?full
```

可以看到类似如下结果:

```
*****
pid:          15342
state:        Running
start time:   21/Oct/2015:22:20:53 +0800
start since:  148
```



```

requests: 238
request duration: 28431
request method: GET
request URI: /forum.php?mod=viewthread&tid=455
content length: 0
user: -
script: /data/www.foo.com/forum.php
last request cpu: 0.00
last request memory: 0

```

各字段的意义具体如下。

- ❑ pid: 工作进程 PID, 可以单独 kill 这个进程。
- ❑ state: 当前进程的状态 (Idle, Running 等)。
- ❑ start time: 进程启动的时间。
- ❑ start since: 当前进程已经运行的时长。
- ❑ requests: 当前工作进程处理了多少个请求, 达到 max\_request 值后会自动被 kill 掉, 防止内存泄漏的长期影响。
- ❑ request duration: 请求时长 (单位为微秒)。
- ❑ request method: 请求方法 (GET、POST 等)。
- ❑ request URI: 请求 URI。
- ❑ content length: 请求内容的长度 (仅用于 POST)。
- ❑ user: 用户 (PHP\_AUTH\_USER)(如果没有设置; 则为 '-')。
- ❑ script: PHP 脚本 (如果没有设置; 则为 '-')。
- ❑ last request cpu: 上一个请求的 CPU 使用率。
- ❑ last request memory: 上一个请求使用的内存。

从结果中可以看出正在处理 forum.php 这个脚本, URI 是 /forum.php?mod=viewthread&tid=455, 这个工作进程存活了 148s, 一共处理了 238 个请求, 这次请求已经耗费了 28431 微秒 (28 毫秒)。如果看到所有的进程都处于 running 状态, 则表示很繁忙。如果经常看到有请求时长比较大的请求, 则表示可能存在问题; 如果看到多个进程都在处理同一个 URI, 则表示这个 URI 的请求可能很多, 或者很慢。

### 3. 内部运行状态分析, 代码级性能

PHP 代码性能是属于研发侧的工作, 不是光靠运维就可以解决的, 不过运维面对的是生产环境, 更有可能比研发先发现隐性问题。运维不会亲自修改代码, 不过运维可以告诉研发, 哪里存在性能问题, 可能是什么原因引起的。所以这里不会具体讲代码的优化方法, 而是讲解如何分析和发现 PHP 代码的性能问题。

#### 1) 通过 xhprof 分析 PHP 代码级性能

xhprof 是 Facebook 推出的 PHP 扩展, 能够快速发现 PHP 执行的内部执行和资源消耗情况。关于具体的安装和配置过程这里不做解释, 有兴趣的读者可以上网查找资料。效果图如

图 7-33 所示:

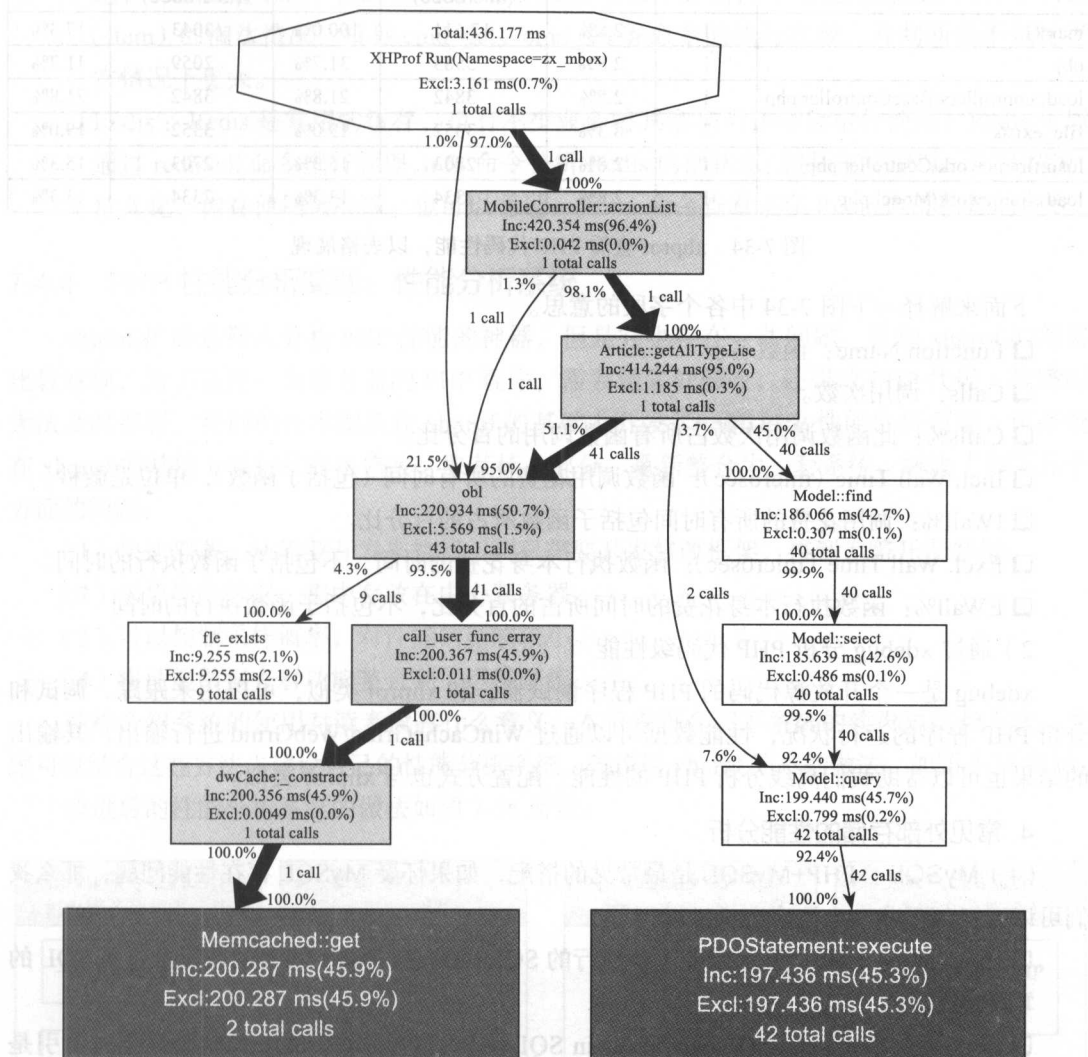


图 7-33 xhprof 查看 PHP 性能, 显示函数调用链

从图 7-33 中可以看出 PHP 代码函数之间的调用关系, 箭头表示调用关系。粗箭头表示最耗时的调用链, 深色背景的是最耗时的函数。百分号表示本函数占了本次调用链的耗时比例是多少。一个函数在调用链上可能被调用多次, 箭头线上的“call”表示调用了多少次。从图 7-33 中可以看出最耗时的是最粗的箭头, 往下看最终是调用了 `memcached::get` 函数, 该函数占了本调用链 45.9% 的时间。

图 7-34 以表格形式展示了另外一种视图, 包括调用次数、比例、耗时情况、消耗内存情况。



Function Name	Calls	Calls%	Incl.Wall Time (microsec)	IWall%	Excl.Wall Time (microsec)	EWall%
main()	1	2.8%	17 611	100.0%	3043	17.3%
obj	1	2.8%	5585	31.7%	2059	11.7%
load::controllers/BaseController.php	1	2.8%	3842	21.8%	3842	21.8%
file_exists	3	8.3%	3352	19.0%	3352	19.0%
load::framework/Controller.php	1	2.8%	2703	15.3%	2703	15.3%
load::framework/Model.php	1	2.8%	2334	13.3%	2334	13.3%

图 7-34 xhprof 查看 PHP 代码性能，以表格展现

下面来解释一下图 7-34 中各个字段的意思。

❑ Function Name：函数名。

❑ Calls：调用次数。

❑ Calls%：此函数调用次数占所有函数调用的百分比。

❑ Incl. Wall Time (microsec)：函数调用花费的所有时间（包括子函数），单位是微秒。

❑ IWall%：调用花费的所有时间包括子函数所占的百分比。

❑ Excl. Wall Time (microsec)：函数执行本身花费的时间，不包括子函数执行的时间。

❑ EWall%：函数执行本身花费的时间所占的百分比，不包括子函数执行的时间

## 2) 通过 xdebug 分析 PHP 代码级性能

xdebug 是一个开放源代码的 PHP 程序调试器，和 xhprof 类似，可以用来跟踪、调试和分析 PHP 程序的运行状况，性能数据可以通过 WinCacheGrind/WebGrind 进行输出，其输出的结果也可以帮助我们离线分析 PHP 的性能。配置方式也与 xhprof 类似。

## 4. 常见外部存储的性能分析

(1) MySQL：PHP+MySQL 是最常见的搭配，如果怀疑 MySQL 存在性能问题，那么我们可以通过如下几个方法来分析性能问题。

❑ show processlist：显示当前正在运行的 SQL 的信息，分析 SQL 的数量及每条 SQL 的当前状态。

❑ explain：取出可疑 SQL 执行 explain SQL 语句，可以分析 SQL 的执行计划、索引是否合理等。

❑ show profile：通过 show profile 命令，可以分析出 SQL 的详细执行情况。

❑ show status/show global status：可以查看 MySQL 内部的各种状态，包括连接、线程、内存使用、数据表的打开数量、SQL 命令执行情况等。like 子命令可以过滤信息，如 show status like “%cache%”。

❑ show variables：可以查看当前生效的变量值，结合 show status 可以查看配置是否合理，这个比查看 my.cnf 文件更准确。

(2) 慢日志 (slow log)：如果是经常性的慢，则可以执行 mysqldumpslow slowlog 文件分析 MySQL 的慢查询日志。

❑ **Memcached** : Memcached 也是常用的缓存组件, 如果怀疑 Memcached 出现性能问题, 可以通过 `stats`、`stats items`、`stats slabs` 等命令分析内存的使用情况, 内存中项目 (item) 的淘汰情况, 每秒 `cmd_get`、`cmd_set` 等命令的执行次数, 并判断是不是比正常情况下更大。

❑ **Redis** : Redis 是常用的缓存, 也有不少业务将其拿来当作存储组件使用。Redis 可以通过 `redis-cli` 命令进行管理, `info` 命令分析 Redis 内部状态: 包括连接情况、主从情况、持久化、内存使用情况等。也可以通过 `monitor` 命令查看正在 Redis 内部执行的指令。

#### 7.4.4 PHP 性能分析实践: 性能分析系统

xhprof 扩展是深入分析 PHP 性能的神器, 但是它也存在一些问题, 比如 xhprof 的部署比较麻烦, 为了监控一台服务器的 PHP 程序, 需要做多处修改, 还得改 PHP 代码, 排障时无法及时部署。我们的技术团队在 xhprof 的基础上开发了一套 PHP 性能分析系统。该系统在 xhprof 的基础上进行了简单修改, 取其核心部分, 重新整合为一套系统, 解决了如下几个方面的问题:

- (1) 快速部署, 从需要开发来执行, 到只需要开发修改框架, 再到无需开发参与。
- (2) 采集性能数据, 集中存放在中心服务器。
- (3) 可以控制采样概率, 对性能的影响较小。
- (4) 将性能数据查看功能整合到了运维系统。

纯粹介绍系统的使用对读者没有什么意义, 本节着重介绍下系统的建设思路和方法, 大家可以结合这些方法去建设自己的性能分析系统。先说说 xhprof 原本的做法, 如图 7-35 所示: 改进后的性能分析系统的做法如图 7-36 所示:

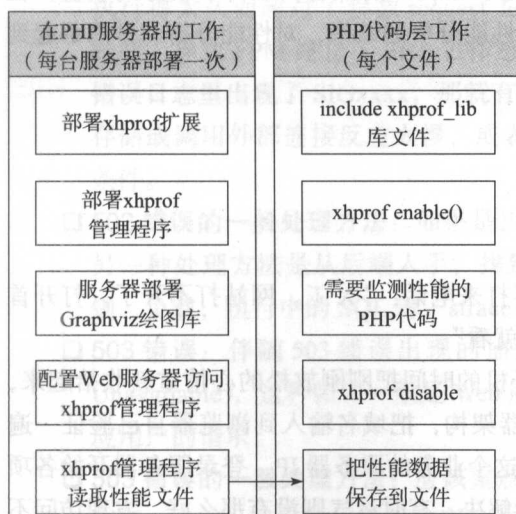


图 7-35 使用原本 xhprof 扩展的必要工作

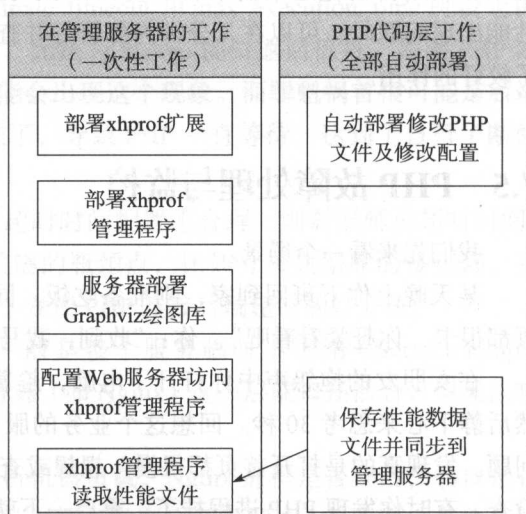


图 7-36 改进后的性能分析系统的必要工作

xhprof 管理程序是一套 PHP 系统代码，展示图（图 7-33）和表格（图 7-34）都是通过这套程序实现的，Graphviz 是一套绘图程序，可以读取性能文件中的调用链绘制调用关系图（如图 7-35 所示）

那么，自动部署是如何做的呢，我们写好了两个 PHP 程序，分别称为 xhprof\_start.php 和 xhprof\_end.php，在为某台服务器开启性能分析的同时将这两个文件发布到目标服务器，并自动修改 php.ini 文件中的选项：

```
auto_append_file =xhprof_start.php,
auto_prepend_file =xhprof_end.php
```

这两个文件将分别执行图 7-35 中右边框的所有工作，而且还能控制采集性能数据的频次。性能分析系统的工作流程如图 7-37 所示：

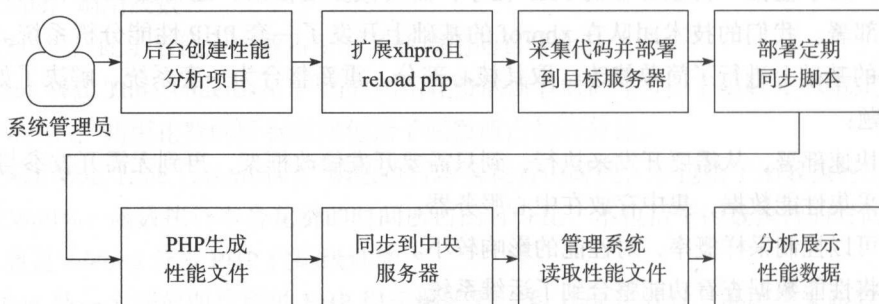


图 7-37 性能分析系统工作流程

这套系统做得比较小，使用起来也比较简单，工作原理如图 7-37 所示。系统的工作量较小，利用自动化运维技术和一些 PHP 技巧简化了 xhprof 的部署，取得了较好的收益，在性能出现问题时，可以在几分钟内部署好并查看性能分析的结果，对性能分析和排障都起到了较好的作用。

## 7.5 PHP 故障处理与监控

我们先来看一个场景。

某天晚上你下班回到家，刚准备吃饭。同事打来电话：“X 工，网站打不开了，打开首页都很卡，你赶紧看看吧”。你：“收到，我马上就看”。

在女朋友的抱怨声中你打开了电脑，趁着开机的时间把刚刚放松的心情重新收拾起来，然后静下心来思考 30 秒。回想这个业务的服务器架构，把域名输入到浏览器自己验证一遍问题。发现真的是打开首页都很慢，想想或查找这个业务的服务器 IP，登录服务器开始各项检查。有时你发现 PHP 进程挂了，重启一下就能解决，有时运气则没有那么好，发现访问不断出现错误 502，排查半天也找不到原因。还得叫上研发、DBA 上线一起分析。等你查完问

题，女朋友已经吃完饭了。

是不是很熟悉的场景？也许你觉得这些好像都很正常，都是这样的过程呀。等等，故障从用户反馈时到解决完已经不知道发生了多久，你如果很熟悉业务则可以回忆起架构，但是如果不熟悉，那就还得找架构图或 CMDB，这资料还得是最新的。如果机器负载看起来还比较正常，你得再静下心来想想问题到底出在哪里了，即使发现是机器负载的问题，你又怎样才能快速处理呢？可是你想过没有，时间就这样在一分一秒地过去，而我们对故障还是有点懵。如果是经验丰富的研发或运维或许还能知道问题大概出在哪里。但如果是运维新手呢，只能找研发一起慢慢排查，而用户是不会等你的，所以你只好眼睁睁地看着用户流失了。大型网站上是不允许出现这种情况的，一旦出现老板肯定要追责，要进行事故定级，如果背上个一二级事故，那么年终奖恐怕得大打折扣。

故障问题跟性能问题息息相关，又有不同之处。性能可以体现为吞吐低和高延时，达到一定的量或突破极限后往往就成为了故障。本节将尝试为大家分析 PHP 故障，介绍快速分析方法和解决方法。

### 7.5.1 PHP 故障分类及处理思路

有经验的运维一般都能够快速定位到问题点，他们接到报障时就已经在心里根据故障的描述来对问题进行了分类，进行准确的分类判断是进行下一步分析的基础，只有准确分类才能一矢中的。大部分 PHP 故障会反映在用户端，用户看到的可能是 HTTP 错误返回码、也有可能表现为卡顿。在没有经过异常处理的网站中，常见的错误码有如下几种：

- 502 错误：打开页面显示的是 502 bad gateway（错误网关）的信息。原因一般是 PHP 执行得太久而超过了参数 `request_terminate_timeout` 和 `max_execution_time` 所设定的时间，导致 FPM 终止了 PHP 工作进程，无法为 `fastcgi` 接口返回信息。如果 PHP 的错误日志里出现了 `SIGxxxx`，那就有可能出现这个现象。而罪魁祸首很可能是后端存储或调用外部连接反应太慢，或者挂了，导致 PHP 一直等待，达到了自行了断的条件。
- 502 错误的一般处理方法：如果是执行超时时间配置不合理，则需要延长超时时间。另一种处理方法是从后端入手，找到关键的瓶颈点，比如分析数据库的慢查询、死锁、负载，执行中的 SQL 等，`strace` 是个发现 PHP 连接后端超时的好方法。
- 503 错误：伴随 503 错误出现的描述一般是整个服务临时不可用（Service temporarily Unavailable），这种错误一般是 Web 服务器（如 Nginx）这一层端口还活着，但无法响应用户的请求。
- 503 错误的一般处理方法：应该重点分析机器负载，Nginx 进程是否存活，端口是否存活，Nginx 进程的工作饱和度，如果以上都正常则应该参照 7.4 节所讲的方法分析 Nginx 状态。

❑ 504 错误：504 错误表示网关超时（gateway timeout），可能是请求量突发，导致 PHP 负载太高，无法为请求分配工作进程，或者达到了连接数限制、某些组件达到性能瓶颈了，导致系统可用性明显下降；影响 504 错误的是 Nginx 的超时设置 `fastcgi_connect_timeout`、`fastcgi_send_timeout`、`fastcgi_read_timeout`。

❑ 504 错误的一般处理方法：还是要从 PHP 层进行分析，如果每个请求的处理时间都是正常的，那么就是整个 PHP 层的资源不够用了，需要扩容；如果是某些请求的时间太长，那么就得找到性能瓶颈点了。

❑ 404 错误：表示页面不存在，静态页面或 PHP 文件被删除。

日常正常运行的网站如果突然变得卡顿，那么出现的故障大部分是性能问题造成的，往往是到了高峰期用户大量涌进网站，导致某些组件达到性能瓶颈。由 PHP 的性能问题所引起的故障比较常见的有如下几种。

❑ PHP 进程不存在：可能由于进程意外掉挂了，再没有起来，比较常见的是请求一直在处理中，达到 `max_execute_time` 时间，管理进程把工作进程干掉了。

❑ PHP 进程响应慢，Web 服务器响应慢，达到超时后，返回 503 错误。

❑ PHP 进程本身慢：这种情况反而不多，只出现在少数特别复杂的框架写的程序中。

❑ PHP 依赖的接口慢：比如请求外部的接口，外部接口很慢等。

❑ 依赖的后端存储慢：这种情况比较普遍，比如 MySQL、Redis、Memcached 出现故障或性能问题。

## 7.5.2 业务监控和故障发现

大部分网站可能是等用户抱怨的时候才发现有故障，而这时不知道已经影响了多少人、出现了多久，这时候老板很可能也已经发现了，运维处理起来会很仓促，压力也很大。好的故障处理是要做到比用户和老板先发现故障，甚至还没有出现故障，而只是一个风险点的时候就把它解决掉。所以故障发现是非常有必要的，故障发现最重要的方式就是监控。大部分故障的原因是性能问题达到了不可容忍的程度，7.4 节已经讲解了很多 PHP 性能分析的方法，只要关注这些性能指标的变化，就可以提前发现故障的苗头。说起来很容易，但是应该如何做呢，首先我们可以回顾有哪些点可以发现故障线索，哪些点需要监控，应尽可能地收集性能数据和用户访问数据，智能监控，关联分析，而不是等到业务完全不可用了时才发现。

### 1. 监控点和监控方法

❑ 业务的 URL：如果用户访问的页面都出现了问题，这是故障最直观的体现，则要加入 URL 监控，设定合理的超时值。可以使用服务器模拟用户访问，或者使用基调、博锐、监控宝等第三方服务。

❑ PHP 状态信息：监控 PHP-FPM 的 PHP 状态（status）页面，查看当前正在处理的请



求、RPM 等，定期获取，并记录到监控系统，设定好阈值，进行告警，多台服务器可以归并告警。

- ❑ 错误日志的数量和比例：监控 PHP 的错误日志和 FPM 的错误日志，还有慢日志。可以通过数量得到错误率。通过日志内容进行双重监控。
- ❑ PHP 访问处理时间：通过监控 PHP 的访问日志得到每个 PHP 请求的响应时间。
- ❑ 业务日志：可以在业务代码中的关键处理点，如果出现错误或特别耗时的操作超时了则打印日志，使得运维对日志做监控。
- ❑ 吞吐量（单位为 rpm，每分钟请求数）：通过访问日志可以得到 RPM，对突发的请求量提前介入观察，并记录下来作为长期统计值。
- ❑ 进程存活和端口：监控 PHP 进程是否存活，监控 PHP 的端口是否存活。
- ❑ Web 服务器的错误日志：Nginx 到 PHP 的 fastcgi\_connect、read\_timeout、send\_timeout 等的超时时间，如果出现超时则写入 Nginx 错误日志，可以监控出现超时的日志数量。
- ❑ MySQL 等的监控：PHP 的性能严重依赖后端的性能，所以后端也要加强监控。MySQL 可监控的项目非常多，具体请查阅 MySQL 的 show status 命令。
- ❑ 性能分析系统：上面都是各种总体的监控，一旦出现故障，排查还是无法准确定位，这时如果有一套性能分析系统，也许就能迅速发现 PHP 的故障点了，从而可以反馈给研发做快速处理，而不是呼叫研发上线慢慢分析和排查并最终处理。
- ❑ 外部接口的性能：跨应用性能问题跟踪，比如脚本调用外部资源失败或超时。
- ❑ 合适的报警机制：根据阈值告警出来之后，可以通过各种渠道发送给运维和研发，预警用邮件，告警用短信、IM、电话、定期出报告等。
- ❑ 业务日志监控：很多 PHP 程序员不太习惯使用日志、也不太喜欢打日志、对监控问题不够重视，容易犯以实现功能为唯一目标的错误。这些对大型互联网来说都是不可取的，需要运维和研发一起配合，做好日志和监控，只有这样才能打造出高性能的 PHP 服务，为海量用户提供服务。

## 2. 监控系统

监控系统是运维的核心系统。我们不可能每次都靠人工上服务器去分析问题，所以好的做法是自动把分析结果存入监控系统中，定期持续地记录数据。需要分析时到监控系统中去查看即可。监控系统可以使用开源的 Zabbix、Nagios 等，这些都是优秀的开源监控系统，大公司往往会选择自建监控系统。

例如，如何把 PHP 状态（status）监控上报到 Zabbix，在 Zabbix 中展示出来。在服务器中定期执行：

```
/usr/bin/curl -s "http://127.0.0.1/php_status?xml" |awk -F'<|>' '{ $2 ~ /^active-processes/ {print $3} }
```

即可得到活动的工作进程数，也就是当前并发处理的请求数。

配置 `zabbix.agentd.conf`:

```
UserParameter=active.processes,/usr/bin/curl -s "http://127.0.0.1/php_status?xml"
|awk -F'<|>' '{ $2 ~ /^active-processes/ {print $3} }', active
```

通过 Zabbix 相关模板，即可展示出来。网上有大量的此类教程，请自行查阅。

### 3. 可用性管理

通过上面所讲的各种方法可以发现网站各个点的性能问题，有没有什么办法可以衡量整个网站业务的质量呢？如果老板问起，最近网站的运行质量怎么样，运维总不能说最近告警少了很多，或者多了很多吧。这里我们引入可用性管理的概念，可用性管理是把业务综合质量通过量化的数据表达出来，并且能够报告给业务方和老板，从而量化业务的质量，和运维工作的价值。

□ **可用性管理**：在长期分析性能数据的基础上形成的一套可用性的标准。例如我们定义首页响应时间在 1500ms 内即认为是正常的；或者 PHP-FPM 工作进程最大达到总进程的 80% 是安全的；如果某时刻慢日志数量大幅增加了则可能是出问题了。前面列出的分析点都可以形成一个指标，业界称服务等级指标 (SLI)。我们可以在团队内部确定几个大家普遍关心的指标，形成服务等级目标 (SLO)，期望达到的性能目标（如响应时间在 1500ms 以内，整个服务的 RPM 达到 500 以上等）。

□ **持续性能管理**：性能是一个长期的、动态变化的过程，应该长期自动采集性能数据，以反映业务的性能变化情况。比如每分钟定期采集请求的响应时间，就能得到一条性能曲线图。上面所说的每一个性能分析点都是可以通过长期采集数据来得到曲线图的。拿着这个指标就能定期汇报说，我们的网站性能下降了或提高了。

## 7.5.3 PHP 故障消除的方法

### 1. 快速修复代码问题

□ **回滚**：如果是业务上线所导致的故障，那么最好的做法就是回滚。当然如果涉及有数据库的修改，那么要回滚将会是很困难的。必须要做出选择。是回滚容易解决问题还是应该直接扛住。

□ **快速修改发布上线**：在做了很多修改后发布上线，如果仅是某些小功能点故障，那么一般是不做回滚，而是快速修改，重新上线。同样，这也是权衡后的结果。

### 2. 弹性调度

PHP 故障有时可能只是因为 PHP 的工作进程处理不过来了，如果仅是 CPU 和内存等资源不够用的原因，可以考虑弹性增加 PHP 应用层的服务器，在架构合理时弹性调度可以实现自动扩容、缩容。



- **前提条件:** 应用服务要求做到无状态化, 不在本机中保存与业务相关的数据, 如图片、用户 Session 等。Session 可以通过分布式存储来管理, 图片等业务生产的内容要放到分布式的文件系统中。
- **实现难点:** 自动扩容或缩容要求自动化程度很高, 资源池的管理, 能分配机器资源, 能自动开通数据库等后端存储的权限, 能自动部署应用程序等。Web 服务器能自动伸缩应用服务器, 如动态修改 Nginx 的 upstream 配置。
- **弹性条件:** 能动态计算整个应用层的可用性指标, 可以是 PHP 的 RPM 或响应时间、或者是 CPU 使用率、网络状况或负载均衡的请求响应时间、健康次数等, 如果达到设定的阈值则进行自动扩容或缩容。

### 3. 柔性可用

当整个线上一下子找不到问题了时, 或者是找到了问题又修复不了时, 无资源池可用时, 那就只能考虑降级服务, 以保证核心业务可以正常提供服务, 对不那么重要的就暂时停止提供服务, 比如一个业务提供了看图和上传图片的服务器, 如果出现了故障, 就要考虑先把上传图片暂时停止, 而保证看图业务的正常运行。柔性可用有很多细节策略, 总体上就是牺牲不重要的, 以保证核心重要功能的可用性。

## 7.5.4 故障分析案例

### 【案例 1】SQL 问题的发现和分析过程

**现象:** 某业务反馈比较慢, 有时报 502 错误。

**分析:** PHP 访问日志 (access log) 看到有个查询列表页执行的时间比较长, 进一步通过 strace 工具分析发现连接数据库和等待时间都比较长, 估计是数据库负载较高。在 MySQL 中通过 show processlist 和分析慢查询, 发现了问题 SQL, 有上千条, 其中几条显示被锁了, 分析问题 SQL, 发现是表索引不合理引起的。

**结论和解决方案:** 是 SQL 写得不合理和表设计索引不合理引起的问题, 加索引和优化 SQL 后即可解决, 通过 MySQL explain 命令来分析 SQL。关于 SQL 优化和 MySQL 优化, 有专门的文章详细介绍, 这里就不展开了。如何事前发现问题 SQL, 是运维和 DBA 的课题, 可以考虑通过 SQL 的审计来实现。

### 【案例 2】调用外部接口引起的故障

**现象:** 访问某页面卡, 经常打不开

**分析:** 研发刚换人, 也不熟悉代码, 代码很复杂, 无法逐行阅读。运维上去看访问日志 (access log), 没有特别慢的情况, 慢日志 (slow log) 中可以看到偶尔有 curl\_exec 的调用出现, 利用 Linux 的 strace 工具分析进程, 发现有连接某个 IP, 但不知道是什么服务器, 不是我们自己的。有可能是第三方的, 怀疑是通过 HTTP 协议调用了第三方的接口, 外部接口慢导致 PHP 进程慢。于是我们通过基于 xhprof 的性能分析系统, 发现 (如图 7-38 所示):

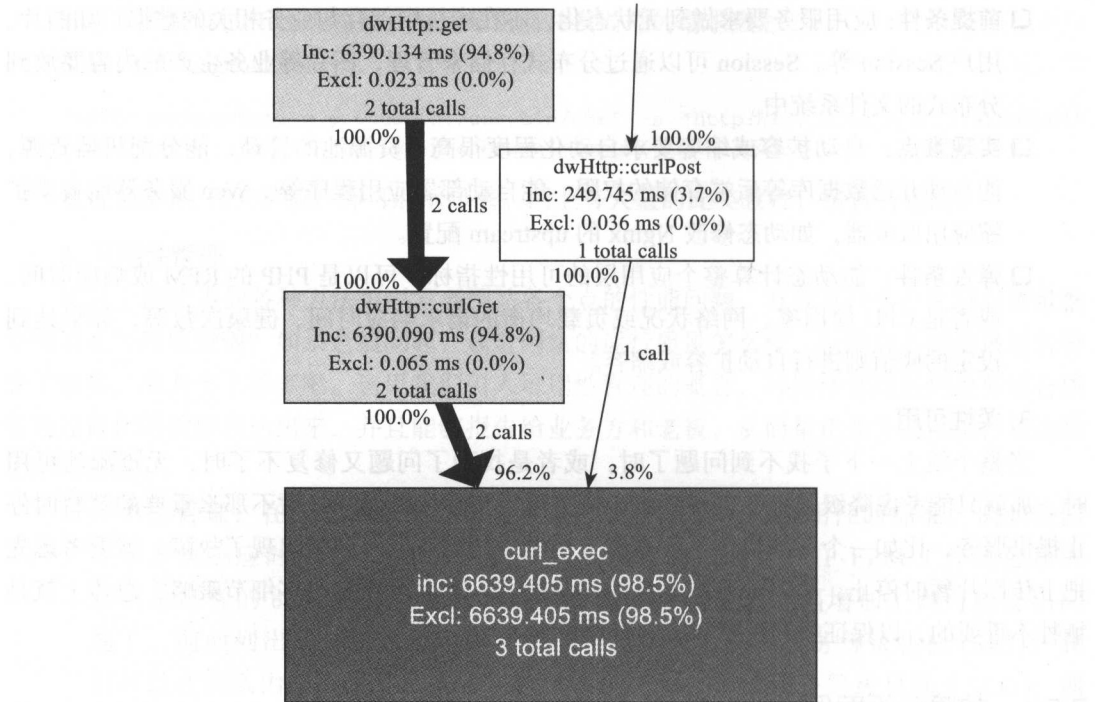


图 7-38 通过 xhprof 发现 PHP 代码性能瓶颈

**结论和解决方案：**curl 访问外部接口占用了太多时间，而且没有设置访问超时时间。让外部接口提供方查找性能问题，同时在 curl\_exec 中设置超时时间为 200ms，如果外部接口不稳定就快速返回失败，而不是卡在那里影响全局。

strace 能在系统层面和 PHP 进程层面发现问题，而不只是单个 PHP 脚本。curl 是访问外部 HTTP 接口的工具，PHP 代码中经常会使用，访问过程需要多个步骤，DNS、连接、等待处理、返回等，时间较长。

### 【案例 3】缓存失效引起的

**现象：**告警系统发现某台缓存服务器的流量突然暴增，达到 900Mbit/s。

**分析：**通过 iftop 等工具发现流量来自于多台应用服务器，缓存服务器端口是 11211，是 Memcached 服务，分析 mc 状态，发现连接数也特别多。

慢日志显示缓存的获取函数较多，access log 显示也较长，但是没有更多的信息。有人认为是缓存失效了，有人认为是缓存服务负载顶不住了，应该增加更多的 mc 服务器负载，也有研发认为是缓存内存不够了，要求加大 Memcached 的内存空间。运维通过 xhprof 分析到有个 PHP get mc 缓存的函数调用 (call) 次数很多。分析 PHP 代码发现几乎每调用一次都要获取一次缓存。从开发经验来讲，如果获取太慢，这已经失去了缓存的作用。和开发沟通后发现是他新增了一个功能，key 只有 10 个，分析后发现 value 都在 40KB 一个，访问量一多，

流量就会暴增。

**结论和解决方案：**很大的 key 而且调用频繁将会导致带宽暴增。这需求有点不太合理，但是业务需要，也无法否定。可通过增加本地缓存来减少读取网络缓存的次数。最后将 PHP apc 的用户缓存作为本地缓存，缓存 30 秒后过期，这样就大大缓解了带宽和调用次数。

## 7.6 小结

很多人对 PHP 有一种固有的印象：它是一种传统的 PHP 开发语言，开发简单，运维简单，似乎很容易就能掌控住。其实在业务量还小的时候，业务如果正常运行，一切似乎都是那么简单、自然。然而在运维几个较大的项目后，我改变了这种看法，其实我们了解的可能只是一小部分，且我们对 PHP 的了解并不像自己想象的那么深入。也有不少人认为 PHP 技术栈的团队无法解决较大规模、高并发项目的性能问题，其实我认为这也许是团队运维建设不足的原因吧。

本章是个人几年来积累的一点点经验总结，希望能对读者在 PHP 运维工作上有所帮助。7.1、7.2 节讲了一些 PHP 认识和开发、架构等较偏原理方面的内容，有了较深的认识和理解之后，对运维工作会有很大的帮助。7.3 节讲了 PHP 进程的部署，PHP 代码的发布，这些在有些读者看来可能无关紧要，但在稍大的团队或多团队的公司里，这些会显得重要得多，提升的不仅是个人的效率，更是提升了整个团队乃至整个公司的效率，减少了沟通、等待和争论的成本。PHP 的性能问题和故障是分不开的，故障处理可能是运维做得最多的工作，7.4、7.5 节讲 PHP 的性能分析和故障处理时，讲到了多种方法用于分析 PHP 的性能问题，以及故障处理的一些思考。

随着云计算、移动互联网、各种海量的服务器规模和业务规模的网站项目的发展，运维所面临的挑战也越来越大，同时也要看到最近几年运维界的发展很快、交流分享也很活跃，各种技术、平台都涌现出来，我认为这对广大运维既是挑战也是机遇。本章是对 PHP 运维技术的一些总结，也融合了对运维技术、自动化、平台化的一些思考，希望对读者有所帮助，这也是我莫大的荣幸。

## 应用系统运行分析

### 作者简介

彭华盛，目前任职广发银行数据中心，是渠道交易应用系统团队负责人，主要负责互联网、中间业务、呼叫中心等渠道交易应用系统运维管理，同时也负责监控等自动化运维建设，对应用生产运维管理、自动化建设等工程有较深刻的见解，有兴趣的读者可以关注作者的微信公众号“运维之路”（微信号：HuashengPeng001）。

随着业务的发展，企业里新建应用系统的数量迅速增加，架构变得越来越复杂，业务对应用系统的稳定性和可靠性的要求也越来越高。在传统企业中，运维部门主要往基础云建设与业务大数据建设的方向发展，而对于承载业务运营开展的应用系统的运行情况缺乏足够的关注，导致应用运维团队疲于被动处理故障应急和善后，以及重复的整改处理工作中。

应用运维团队除了多干活以外，还面临这样一个窘境：老板认为应用运维做得不够细致，保障不力；业务认为应用运维不给力，系统不稳定；开发觉得应用运维不懂应用，只会做操作性的工作。

为解决应用运维的困境，我们开展了一系列应用运维的优化工作，包括操作性工作的自动化建设，并探索应用运维向运维开发、运维分析转型。本文是从运维分析的角度来讲解应用运维团队在应用系统运行分析方面所做的自我解放，从被动式向前瞻性运维方式的转变。本章的写作目的就是帮助应用运维人员顺利完成这种转变，这里主要从应用系统运行分析的角度展开来讲。

应用系统运行分析具有以下特点：

□ 涵盖系统资源、数据库、中间件等标准化的性能容量数据分析。

□ 涵盖应用交易级（交易量、耗时、成功率、响应率等）标准化的性能数据分析。

□ 提供各应用系统自身业务特点的业务专家式数据分析的数据采集、数据建模的自动化支持。

□ 提供可视化、数据采集、数据建模分析、数据归档、数据访问接口的系统自动化支持。

□ 提供可订阅式的可视化报告，供业务和运维管理团队决策之用。

运行分析所涉及的运营分析系统架构

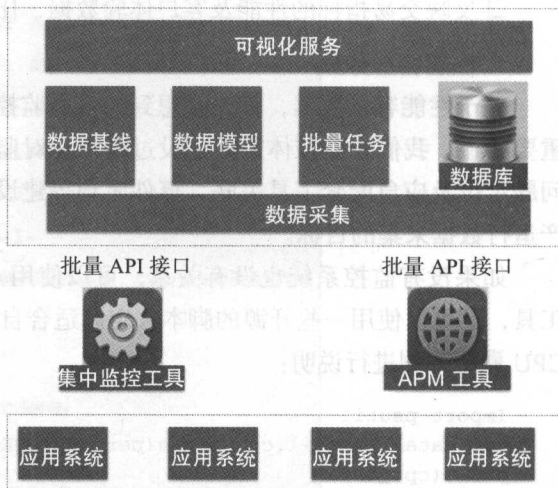


图 8-1 运行分析架构图

如图 8-1 所示：

## 8.1 分析模型

在制定运行分析方案时，不仅要考虑运行分析需要分析什么信息，还需要考虑如何减少方案的实施推广难度。为了便于推广，建议将数据的采集、保存和标准化分析交给机器去处理，运维人员尽量只负责观察和分析数据，为此在应用运行分析的过程中，我们定了如下 3 个原则。

□ 能自动化的则用自动化来实现：分析涉及的数据采集、加工消费、可视化都可由自动化实现。

□ 解决主要的问题：评估应用运维成熟度的角度有很多，鉴于各应用系统的逻辑不一致，运行分析采用 2/8 原则，主要进行性能、容量、客户体验的分析。

□ 简化分析方法：找出运行分析指标的共性，过滤不同应用系统的差异性，将分析方法原子化。

运行分析是一个长期实施的过程，仅仅依靠运维人员经验的分析很难保证运行分析的质量，因此需要将经验转化为机器分析模型。在分析模型方面，主要是对不同类型系统的分析方法进行分解，以得到一些共性的分析方法，即将经验总结出来的分析方法进行标准化的过程。以下将简单介绍其中的几个模块。

### 8.1.1 数据采集

要分析应用的运行情况，首先需要获得反映应用系统情况的数据，因为运行分析主要是解决性能、容量、客户体验的问题，建议从以下两方面定位好数据来源：



- 反映系统软件运行情况的容量性能数据，比如 CPU、内存、数据库事务等。
- 反映交易指标的性能及客户体验数据，比如交易量、耗时、响应率等。

### 1. 性能容量数据

关于性能容量数据，首先会想到的就是监控系统，监控系统的确是性能容量数据采集的重要渠道。我们在监控体系的建设过程中，对监控体系的定位除了生产运行事件报警、辅助问题定位及应急配套工具关联、事件驱动等建设目标，还定位监控体系中的监控工具作为生产运行数据采集的目标。

如果没有监控系统也没有关系，可以使用一些简单的性能工具，或者使用一些 NMON 工具，还可以使用一些开源的脚本去定制适合自己的系统资源数据，下面以 Python 获取系统 CPU 资源为例进行说明：

```
import psutil
cpu_data = psutil.cpu_times(percpu=True)
print(cpu_data)
# 以上的 Python 脚本可以输出 Linux 操作系统下 CPU 的使用率情况
```

通过对以上数据的分析，可以得到 CPU 的 SYS、USER 等进程使用的资源占比，可以通过 IDLE 看到系统 CPU 空闲的百分比。

### 2. 交易性能数据

交易性能的评估指标有很多，比如交易平均处理时间、交易成功率、交易失败率、指定时间范围内的交易笔数等。对于这些数据的获取，不同的应用系统有不同的方法，比如数据库层面、应用日志等可以以入侵应用系统的方式主动获取，也可以采用类似于旁路 APM 等工具来获取（比如上海天旦的 BPC 工具），以下简单介绍这两种数据获取方式。

第一种方式：修改应用系统代码，由应用主动将性能数据记录在数据库或日志中。以数据库层面为例，推荐在交付运维部门前，在应用系统中提供一个数据运维流水表，通过这个数据流水表来获取交易的概况，以下是其中一个较为通用的数据运维流水表结构，如图 8-2 所示。

图 8-2 所示的数据运维流水表获取了交易标识、交易时间、处理状态、上下游系统关系、具体交易类型等信息，有了这些信息不仅可以知道待分析的应用系统的交易性能情况，还可以看出是什么交易、哪个关联系统影响了整体性能。当然，如果应用系统没有设计此类数据运维流水表，也可以根据实际情况收集类似的数据，但最好能够获得一笔交易的处理时间、状态、下游系统这 3 个关键数据，进而通过这 3 个数据定位一笔交易的处理性能。数据运维流水表的思路同样也适用于文本日志的设计，比如单独记录满足条件的应用日志。

主动在应用代码中注入数据运维流水表或文本日志的方案，还依赖于开发团队的支持力度。在开发团队的支持力度相对较弱时，建议优先在总线类应用系统中注入，主要是因为总线类的节点在企业内部制定了规范的接口标准，总线类节点往往能够比较方便地获得成功率、耗时、交易量波动、响应率等数据。

流水号：唯一标识一笔交易流水的 ID。
上游系统流水号：上游系统流水号，配合上游系统排查问题所需要的关键字。
下游系统流水号：下游系统流水号，配合下游系统排查问题所需要的关键字。
开始时间：交易到应用系统的时间。
结束时间：下游系统返回处理结果的时间。
上游系统渠道：上游系统的渠道标识。
下游系统渠道：下游系统的渠道标识。
实际交易处理渠道：下游系统可能是网关等系统，实际交易处理渠道记录最终处理交易的系统。
交易类型：交易类型。
交易状态：交易是否成功、失败或超时等。
交易返回信息：下游系统返回的处理状态。
交易关键信息 1：预留信息，可以通过交易关键字查到交易，比如证件号、账号等。
交易关键信息 2：
交易关键信息 3：

图 8-2 数据运维流水表结构

但是需要记住一点，无论是数据库还是日志，都要通过异步方式来实现。

第二种方式：借鉴基于网络旁路 APM 工具抓取应用性能数据。这类 APM 工具的工作原理主要是在网络层面部署探针抓取应用报文，并将应用报文发送到集中的应用工具，通过对报文数据进行解码、清洗、合并等操作，获取以下数据信息。

- 交易报文类的信息：返回码、交易码、操作码、成功率、渠道标识、交易响应率、交易笔数、响应时间、流水号、金融交易里的金额、账户等关键数据。
- 网络通信信息：丢包、TCP 重传、带宽、连接成功率等。
- 系统软件信息：数据库、中间件使用响应、并发、连接数情况等。

APM 工具往往可以提供实时数据或批量接口，数据来源更加稳定，且网络旁路抓包的方式不会影响应用性能。

### 3. 数据采集解决方案

我们利用监控工具解决了上面提到的容量性能与交易性能数据来源，采集方案如图 8-3 所示。

常见的应用架构主要可以分为 Web、队列、应用、缓存、数据库这 5 层，每一层均由负载均衡或主备方式部署的服务器节点组成，服务器节点可通过传统专业的监控工具来采集涉及资源类、系统软件运行指标等的的数据。除了要获得每个节点的运行数据之外，还要获得每



个节点之间的运行情况，在这里我们选用基于网络旁路抓取报文的 APM 工具来获取各个点之间的数据。

通过传统的监控工具与 APM 工具，我们获取了点、线形成的二维平面的运行数据，这种组合方式，就如同我们的 GPS 地图软件，传统监控工具获取每个坐标站点的情况，APM 工具获取站点与站点之间是否塞车情况的数据。

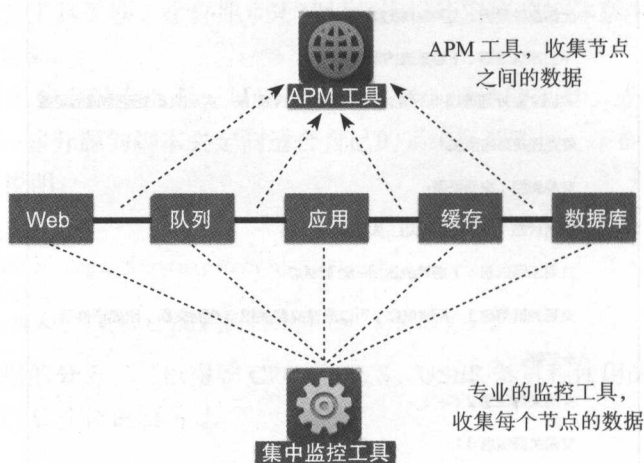


图 8-3 数据采集技术方案

### 8.1.2 数据模型

在应用运行分析的过程中，不同应用系统的架构和业务逻辑均不一样，需要通过标准化的数据分析模型简化分析方法，主要建立了以下几类数据模型：

- 系统资源数据模型。
- 数据库数据模型。
- 交易性能数据模型。

#### 1. 系统资源数据模型

对系统资源数据模型的分析，需要将日常分析系统资源的经验指标化，转化为分析策略，由分析平台以定时批量的方式，对一个应用系统一段时期内的资源使用情况数据，按预指定的策略进行统计分析，并输出可供决策的报表数据。

下面仍以 CPU 为例，CPU 数据通常有以下 3 项需要关注。

- %user：显示用户进程消耗 CPU 的时间百分比。
- %system：显示系统进程消耗 CPU 的时间百分比。
- %idle：显示 CPU 处于空闲状态的时间百分比。

我们在日常实时分析过程中，总结出以下经验。

□ %usr+%system < 70%, %idle > 30%: 较好。

□ %usr+%system > 80%, %idle < 20%: 报警。

□ %usr+%system > 90%, %idle < 10%: 糟糕。

在运行分析的过程中,还需要结合应用系统的特点对一段时间内的趋势进行分析,比如应用系统日终备份或清算等操作,CPU使用率增大,系统会繁忙;业务查询一个报表的查询功能占用CPU的资源可能会在短时间内有一个高峰值;应用系统功能设计不合理,导致功能的执行会消耗大量的系统资源;这些性能数据不能准确反映出系统资源是否满足现有应用系统的性能要求。为此需要根据不同的业务系统制定一个时间段的分析策略,下面为一个实时交易系统制定一个策略(策略可根据企业性质的不同进行定制)。

抽取周一到周五工作日业务高峰期(9:30 ~ 11:30)的CPU性能数据,以15秒为采样间隔。连续20次采样出现CPU性能超阈值(%usr+%system>80%或%idle<20%),则当天存在潜在性能瓶颈,这个潜在性能瓶颈可能由一个正常的操作引起并会在30分钟后得到释放,所以出现一次并不会马上输出CPU资源容量不足的建议。只有当连续3天出现上述情况的潜在性能瓶颈时,才会输出容量不足的建议。

以上的模型中,高峰期的时间段、连续的天数、每天连续出现的次数和指标数据都可以根据运维人员的需要进行调整。

在策略制定之后,分析平台定时执行对以上数据的分析,将扩容的建议与原因作为一个运行分行模块报告,先由应用运维确认是否存在应用缺陷所导致的因素,如非应用缺陷所导致,则提交到基础设施团队,由基础设施团队评估扩容事宜。

类似资源层面的指标,还包括内存、磁盘IO、磁盘空间使用、网络信息的使用情况,相应策略的制定需要有一个使用调优的过程。

## 2. 数据库数据模型

关系型数据库仍然是目前应用系统最重要的数据库,对数据库的分析可以直接反映系统的性能情况。

(1) 对数据库进行实时数据分析,往往是通过一些数据库工具对数据库进行监控分析来实现的。下面以DB2为例,我们往往使用Event Monitor和Snapshot,两者都可以用于实时采集并分析数据库的使用情况,例如数据库缓冲池(buffer pool)的使用状况、即时的数据库locking状态、SQL语句的信息等。

选择任意一个工具都可以,两者都提供如表8-1所示的信息:

表 8-1 数据库运行指标类型

事件类型	提供的信息
Deadlocks	只提供参与到 deadlock 的应用名
Statements	SQL 语句开始/结束时间、CPU 使用情况、动态 SQL 语句文本、语句执行结果,以及 fetch 记录数量等信息
Transactions	UOW 开始/结束时间、CPU 使用情况、locking 及 logging 等信息

(续)

事件类型	提供的信息
Bufferpools	各 buffer pool 的预读、page 切换及直接 I/O 等信息
Tablespaces	各 table space 的预读、page 切换及直接 I/O 等信息
Tables	各数据库表的读 / 写字数

通过在高峰期对数据库锁、SQL 语句等进行分析，不仅可以得到一些数据库参数配置的建议，还可以得到一些应用层面的调优建议，比如对 Statements 项的分析可以得到高峰期执行时间最长，资源消耗最大的数据：

- ❑ 执行次数 top20 的 SQL。
- ❑ 消耗时间 top20 的 SQL。
- ❑ 返回记录数 top20 的 SQL。

有了这类 SQL 语句信息，就可以通过 DB2 的建议工具生成 SQL 优化建议，通过新增一些数据库索引来提高应用的系统性能。

(2) 通过数据库的数据统计信息可以得到数据量大的表，任由数据库表数据量增大而不做数据迁移或清理，将会导致数据库性能瓶颈的出现。仍以 BD2 为例，可以通过以下 SQL 语句查到一个数据库中各表的数据（该表的数据需要在重整后才能得到最新的统计信息）：

```
select a.card as rows_num,b.colcard as Cardinality,create_time,alter_time,stats_time from syscat.tables a left join syscat.columns b on a.tabschema=b.tabschema and a.tabname=b.tabname order by a.card desc
```

上述代码各字段具体说明如下。

- ❑ rows\_num：数据行数。
- ❑ Cardinality：数据基数。
- ❑ create\_time：表创建时间。
- ❑ alter\_time：表修改时间。
- ❑ stats\_time：信息统计时间。

通过以上 SQL 语句可以导出数据量排行前 10 位的数据表，提供给运维人员以重点分析是否需要对该 10 张数据表进行数据迁移，以减少存量数据。

(3) 另外，数据库索引在实际调优过程中是最直接、最快速、最有效的方法，索引的分析有极为重要的意义。同样，数据库也为分析数据库索引提供了丰富的方法：分析索引的引用次数，删除无用的索引；也可以对索引是否重复进行分析，比如索引 1 (id,name)，索引 2 (id)，这里的索引 2 即为重复多余的索引。

### 3. 交易性能数据模型

衡量一个交易系统的性能，还需要从交易层面进行分析，以下将介绍几个常见的交易性能数据模型。

### (1) 交易笔数

交易笔数的分析,主要是通过在指定时间内对交易发生的笔数进行分析,建议在交易笔数的分析中定义如下4种模型:

#### 1) 按笔数分析的情况。

交易总笔数:  
交易成功笔数:  
交易失败笔数:  
交易异常笔数:

对交易总笔数进行分析,可以得到系统每天或每天的高峰期指定时间段的处理的交易总笔数、交易成功率、交易失败率、交易异常比率(交易异常主要是指交易超时,有别于交易失败,交易失败可能是正常的,比如余额不足导致的交易失败)。通过连续一个月的数据分析,运维人员可以查看交易的分布,分析交易是否呈上升的趋势,系统处理交易的成功率是否下降等情况,再进行深入分析是什么因素引起的,最终提出优化建议。

#### 2) 按渠道分析的情况。

渠道:  
交易总笔数:  
交易成功笔数:  
交易失败笔数:  
交易异常笔数:

上面是对应用系统的整体处理能力进行分析,但由于现在的交易系统往往不是孤立存在的,关联系统的性能下降也可能导致应用系统的性能下降,比如:分析应用系统的并发连接数为100,如果下游某个系统交易缓慢,每笔交易都是30秒才返回,那么当该下游系统交易数量变大时,可能会导致并发处理线程不能及时释放而无法处理更多的交易。所以还需要按渠道分析的模型,该模型可以看出哪些下游系统处理能力比较差,是系统整体处理笔数模型的一个补充。

#### 3) 按交易码分析的情况。

交易码:  
交易总笔数:  
交易成功笔数:  
交易失败笔数:  
交易异常笔数:

交易码分析的情况则是从某个功能设计的角度来分析,比如:如果某几个交易码的失败率明显高于其他系统,则有可能是该交易设计本身有问题,或者该功能有缺陷,失败率高。

#### 4) 按交易返回码分析的情况。

错误码:  
交易总笔数:  
交易成功笔数:

交易失败笔数：

交易异常笔数：

交易返回码是对一些交易渠道处理结果的反映，例如：如果有大量的交易码都是通信超时的，那么可能是某个下游系统的处理能力有问题所导致的。这样的分析模型可以帮助发现交易异常的类型是性能问题，还是功能设计问题。

## （2）交易平均耗时

1) 系统整体交易处理笔数：对应用系统整体交易处理耗时进行分析，可以看到当前系统的平均处理能力，再通过一个周期的交易耗时来进行分析，可以看到平台整体的处理能力是否下降。例如：一个应用系统在上线初期，虽然交易性能不高，但由于存量数据不多，一般不会出现性能瓶颈。当存量数据越来越多时，系统每笔交易的平均处理性能就会显现出来。

2) 按渠道、交易码统计交易耗时：类似于上面交易笔数的分析模型，对交易耗时的分析，除了要对整体交易进行分析以外，还要分渠道、交易码进行分析。这两个模型可参考上面交易笔数的模型进行设计。

## （3）异常交易笔数

单独对异常交易建立一个分析模型，因为在实际的应用性能分析中，除了交易成功、交易失败以外，还有一些无反馈交易处理结果的数据，这类交易可能导致一些单边账交易，不容易被发现，但其出现后影响又会比较大，比如在呼叫系统中如果 IVR（自助语音）交易超时，那么该交易将直接转接到人工坐席；如果异常交易过多，就需要更多的坐席支持，如果坐席人力不足则会导致人工坐席满线而无法提供服务的情况。故有下面所述的这个分析模型。

## 4. 数据模型解决方案

建立数据模型并非将数据做一个简单的汇总和展示，而是从当前运行的数据中发现潜在的运行问题。那么，如何判断当前运行数据是否存在潜在问题则十分关键。在这里我们主要是建立运行基线，通过将当前运行数据与运行基线进行偏离分析来开展工作。

所谓运行基线即根据上一分析周期历史运行数据进行采样，并按一定的周期计算一个平均值，作为这个时点的基线，图 8-4 是运行基线的建设方案。对各应用系统建立工作日和节假日的系统基线，每一条基线由 24 个基线值组成，每个基线值是某一项运行指标数据模型在这个时间周期内数据采样的平均值。在实际运行过程中，我们发现每个小时的基线点都出现了一些分析误差，比如每天晚上凌晨 1 时到 6 时之间，实际发生的交易数据很少，波动性很强；或者每天凌晨 3 时 50 分至 4 时 15 分之间因为批量定时任务导致数据库使用性能突增，3 时与 4 时的数据波动较大。为此，一是要去除干扰的运行数据，二是将基线粒度进一步缩小，计划后续在工作日与节假日的基础上，再细化为星期线，基线值由原来的 1 小时细化到 10 分钟。



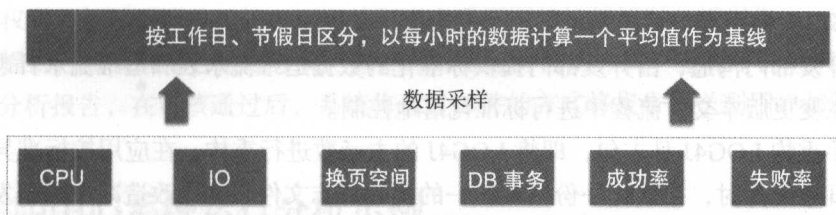


图 8-4 运行基线技术方案

有了基线，就有了系统运行好坏的参考值，接下来要制定策略以定位什么是好，什么是坏。为了减少误报率，主要从以下三个方面进行处理。

- 按偏离次数：以 CPU 为例，我们将 CPU 值在基线上方超过 25% 的点取出，根据点的多少判断是否存在某些时点出现了性能上升的问题，需要应用运维人员分析对应时点的运行情况。
- 按整体偏离趋势：仍以 CPU 为例，我们根据当前运行点的指标与基线进行对比，查看两者正负比率，如出现当前运行点大部分均偏离基线上方，则整体运行性能有下降的趋势，需重点评估是否需要扩容。
- 清理干扰数据：主要是在分析过程中，针对波动性很强的数据进行清理，比如因为交易量少而导致的波动性问题就不在运行分析范围之内。

## 8.2 运行分析平台建设

通过前面所讲数据采集与数据模型的建立，数据分析的基本模块已经实现，接下来需要考虑如何将运行分析推广下去。运行分析平台架构如图 8-5 所示。在运行分析的推广过程中，可能会遇到如下这些问题：

- 需要手工采集运行数据，且数据收集耗时很长。
- 平均耗时 120 毫秒是正常的吗？无法快速找到上个月的分析情况进行对比。
- 分析的报告无法及时提交给决策层进行分析。
- 缺少一个可视化的系统。

本节主要介绍数据的采集、分析、决策，以及数据可视化平台的建设方案。

### 8.2.1 数据采集接口

在数据采集接口方面，我们最初考

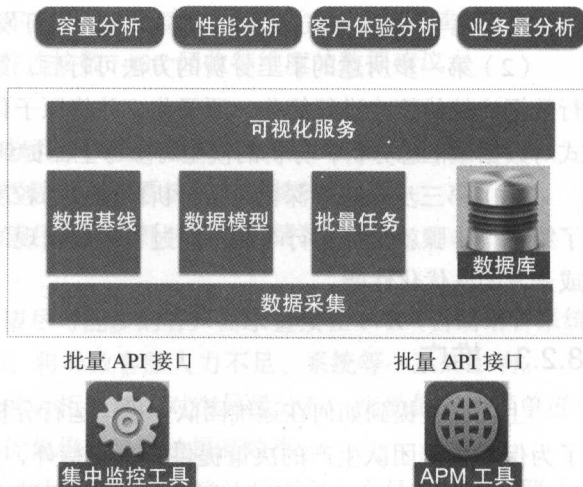


图 8-5 运行分析平台架构

虑由开发部门支持运维，将应用系统的运行数据主动输出成标准化的数据格式：

- 与开发部门沟通，由开发部门提供标准化的数据运维流水表和运维流水日志，运维部门在变更版本交付流程中进行标准化落地控制。
- 着手重构 LOG4J 日志包，即将 LOG4J 的主函数进行重构，在应用按标准异步写入应用日志文件时，格式化一份格式统一的运行日志文件，这个改造减少了开发对应用代码的修改，同时无须安装其他日志代理。

在实施过程中，我们发现由开发部门配合运维部门修改应用旁路，向运维输出有关应用运行情况的数据，阻力会比较大，所以我们分成如下两个步骤进行：

第一步是挑选节点总线类的应用系统进行改造，因为是挑选个别应用系统来进行改造，开发部门没有很多推脱的理由。选择总线类的应用系统主要是因为这类系统接收了大量渠道系统的交易，并负责转发到关联系统，在总线类的应用中可以获得各渠道的交易处理时间、交易处理情况、关联系统返回的情况等信息。也就是对这类系统进行改造所获得的数据，可以用来分析各渠道的整体运行情况，能起到事半功倍的作用。

第二步是我们发现基于网络旁路抓包的 APM 工具可以实现无须改造应用就能获得交易处理时间、成功率等数据，这些数据虽然不能像应用在开发阶段注入运行数据时那么精细，但是采用旁路抓包的方式实施落地更高效，而且在性能分析方面也能满足 80% 的需求。

### 8.2.2 数据分析模块

关于数据分析模块，就如上文提到的，建立运行基线，拿当前运行的数据与运行基线进行比较，通过偏离度与偏离数量判断运行的情况。在建设数据分析模块时，应采用如下几个步骤：

(1) 手工分析数据，发现运行问题，并将分析到的运行问题反馈给应用管理员，由应用管理员进行分析，最终统一用分析数据反推开发中心进行优化，比如刚开始的按渠道分析，将交易单笔平均耗时超过 2 秒的交易渠道发给开发团队，督促开发进行优化。

(2) 第一步所述的手工分析的方法可行后，第二步就是将数据分析的数据采集范围、运行数据比较的策略进行简化、原子化，并将原子化后的操作实现自动化，通过定期批量的方式对数据做汇总分析，分析的模型可参考上面提到的基线比较的方式。

(3) 第三步是持续深化运行分析的深度与粒度，第二个步骤主要是整体的运行情况，到了第三个步骤就是抓运行毛刺，通过将毛刺出现的时间、性能表现集中展示出来，督促运维或开发团队优化性能。

### 8.2.3 推广

上面已经提到如何在运维团队中推广运行分析工作的开展，这还不够，运行数据分析除了为保障运维团队生产的决策提供数据支持外，还需要推广到 IT 管理团队及业务团队。为了更好地推广，需要强化数据分析结果的可视化效果，例如如下两个方面。



(1) 权限：控制不同的机构、不同的用户查看不同的运行情况分析报告。

(2) 提供订阅式的报告推送方式：用户可以通过订阅的方式，查看不同的应用系统或场景的运行分析报告，在审核通过后，系统将通过邮件的方式将报告发送到用户邮箱中。

## 8.3 呼叫中心系统运行分析示例

本节主要通过介绍呼叫中心系统的运行分析来回顾上面提到的运行分析技术方案。呼叫中心系统的运行状态主要包括话务、坐席两部分，其中话务部分又包括网关、自助语音、录音、多媒体等十几个子模块；坐席部分则主要包括坐席应用、运管、报表、网关等模块。整个应用系统涉及150个操作系统分区，如此多服务模块的运维管理必须进行前瞻性的运行分析才能提前发现问题，减少问题。

运行分析主要如下几点：

- 根据运维及业务的需求，对运行分析的服务器范围及交易分析指标做加法与减法。
- 根据业务需求增加个性化的业务分析方向。
- 根据分析的数据问题做进一步分析，并转化为后续优化计划的决策依据。

### 8.3.1 确定分析方案

针对呼叫中心系统的运行分析情况，我们与业务部门进行了沟通，确认了客服满意度、话务量、坐席菜单响应率是业务关注的几个重要指标，同时业务部门发现近期人工坐席的话务较以往有所增长，希望知道问题的原因。这个过程就是根据业务的需求做减法与加法。

针对上述沟通情况，我们制定了如下分析方案。

(1) 资源类指标。

- 系统资源分析：分析系统CPU、内存、IO的资源使用情况。
- 数据库性能分析：分析高峰期数据库索引、锁、大数据表等维度的数据建议。
- WebSphere 中间件分析：分析中间件连接数、GC回收等维度的数据建议。

(2) 应用性能指标。

- 交易平均耗时分析：交易平均耗时分析可以整体感知平台的处理能力概况。
- 按渠道统计交易平均耗时：发现是否涉及关联系统耗时过长的情况。
- 按交易码统计交易平均耗时：发现某几笔交易耗时异常的情况。

(3) 细化的交易级指标。

- IVR 超时交易转人工分析：业务希望尽可能多的客户需求直接在IVR（自助语音系统）中完成交易，如IVR交易超时过多，将导致客服人力不足、系统等一系列影响。
- IVR 交易量情况分析：IVR交易量的分析主要是对交易量分布、交易最多的菜单进行分析，通过对交易最多的菜单进行优化将会最快地提高效率。
- 按交易码统计成功率：发现交易的成功率情况，以确认是否存在交易设计的问题或交

易异常的情况。

(4) 业务指标：业务指标个性化较强，主要以人工分析为主。

□ 呼叫中心客服满意度分析：从业务最关心的客服满意度指标进行分析，包括总进线量、总接通数、接通率、人工转接数、排队返回主菜单、20秒水平、平均应急速度、平均通话时长、话后处理时长、平均整体时长、坐席待机率几个指标。

□ 话务量情况分析：分析话务量在一个月内的分布情况，发现话务量高峰期的特点，以协助业务人员安排座席人员班次的人数。

□ 重复来电情况分析：发现是否有恶意来电攻击的情况。

□ 话务平台录音丢失专项分析：分析录音丢失的情况。

### 8.3.2 问题分析案例介绍

在分析过程中，以业务部门需要分析的关于近期人工话务量增长较多的情况为例，分析的情况如下。

□ 近一个月话务量的分布有如下特点：星期五的话务量相比其他几天增长了20%，本月的话务量相比上月话务量有明显增长的趋势。

□ 本月下旬25日开始每天均出现IVR超时的情况，IVR超时情况集中出现在上午9时到11时。

□ 交易整体耗时方面，从本月25日开始，每笔交易的平均耗时由原来的150毫秒上升到250毫秒（500毫秒将对客户体验产生明显影响）。

□ 客服满意度分析中排队返回主菜单数、坐席待机率指标均有变差，问题同样集中在9时到11时，存在坐席不足爆线的情况。

从以上三个分析点可以看出：在话务量未明显增长的情况下，IVR在上午9时至11时出现了IVR交易超时情况，这些超时交易全部转到人工坐席，人工坐席出现爆线情况。初步结论是：业务部门需要先应急扩大9时至11时的坐席人员数量，技术方面需要进一步分析9时至11时是否有一些特别的交易导致整体超时的问题，重点评估24日晚是否有因生产变更而带来一些业务功能，这些业务功能主要在上午9时至11时发生交易。

经过分析最终发现了问题的原因，24日晚投产了一个查询前一日客服处理情况的实时报表，业务人员往往会在次日上班9时后进行查询。由于该功能需要在多张上亿条数据的表中关联查询数据，效率低下，因此影响了整体的性能。

所以运维人员针对IVR人工话务量增多方面的分析结论附上问题原因之后，提出了如下建议。

(1) 近期工作建议：

□ 因该查询功能是T-1日数据，建议开发团队将该功能设置在凌晨批次生产报表，减少在业务高峰期进行统计类的查询业务。

□ 建议运维人员对IVR超时交易进行交易级监控配置，当IVR出现交易超时提供监

控报警。

- 建议对生产实时表数据过亿的表格进行历史数据迁移,以保持实时表的高效。
- 建议业务人员对爆线情况进行现场监督,及时调配好班次人员。

(2) 中期工作建议:

- 建议将查询报表类的模块单独拆分为独立的应用服务。
- 建议将查询报表类的数据库进行配套的拆分。
- 建议对数据库中的数据实现读写分离,优先考虑增加一个复制库,在业务空闲期间定时生成批量报表,对于需要实时查询的数据采用读取复制库的方式;同时研究读写分离中间件。
- 建议对报表类交易设计降级开关,当实时交易发生异常时,要能够快速暂停报表类交易。

## 8.4 小结

上述应用运行方案仍在实施中不断进行完善,后续主要的优化方向是:

- (1) 通过应用运行分析的开展,全面转变运维人员的工作思维,由操作向分析进行转变,由局部到整体进行把控,由被动应急向主动出击进行转变。
- (2) 为应用系统的资源、性能运行情况建立一个运行基线(优、正常、差),为系统管理提供容量决策支持,反推开发团队优化应用性能,推动应用架构改造,进一步保障系统稳定运行。
- (3) 通过大数据技术的引入,提高运行分析效率,将运行分析由 T+N 向 T 日实时分析的方向发展,为监控自动化工具提供预测性的事件预警等,完善运维自动化体系的建设。
- (4) 提高应用运行分析的深度,将运行数据与业务运营活动的开展、客户体验等方面进行数据关联分析,挖掘影响业务开展的功能设计或应用性能因素,促进业务开展。

## 虚拟化中存储配置典型场景：启动风暴

### 作者简介

蒋迪，上海沃帆信息科技有限公司资深虚拟化基础架构工程师，从事云平台研发工作，主要包括 OpenStack、oVirt 的虚拟化、网络、存储组件及其在桌面云领域的应用。涉及虚拟化产品应用（云桌面、云服务器），云平台架构与相关开发（自主产品与开源软件），嵌入式系统与设备（ARM 架构），分布式存储（GlusterFS）。多次参与银行与教育领域的私有云架构设计与实施。

启动风暴是私有云桌面中经常会遇到的问题，不管是开源平台还是闭源平台，或多或少都会遇到此类问题。本章将以 oVirt 为基础平台介绍其平台存储配置，并针对这种配置下的桌面云启动风暴进行调优，然后在本章末尾处会介绍类似于杀毒风暴的解决办法，希望通过本章的介绍，读者在解决此类问题时能够尝试从不同的角度进行思考。

在启动虚拟桌面时，虚拟机系统内部会产生远高于正常运行状态时的 CPU 负载与 I/O 请求，当同时启动的虚拟机达到一定数量时，就会对服务器造成极大的压力，这种情况一般称为“启动风暴”。在私有桌面云中，尤其是教学环境中，会定时定量集中启动一批桌面，如果启动时服务器的 CPU 能力或存储能力未能满足其需要，那么就会对全部桌面的启动造成影响，甚至假死。类似于启动风暴对存储 I/O 会造成压力的场景还有登录风暴、杀毒风暴（扫描风暴）等。

启动风暴的发生条件并不确定，因为它与服务器的 CPU、存储、系统配置及虚拟机的参数直接相关，所以改善启动风暴大多是从这些相关方面直接入手。

以我个人的经验和客户反馈的情况来说，其中存储的改变对启动风暴的影响是最大的，

而针对存储的优化则可以从最基本的存储性能及其存储结构方面进行调整。

本章将尝试模拟桌面云的一般场景，启动相当数量的虚拟机来重现启动风暴，尝试使用分层存储的方法处理之，然后在现有方法的基础上提出一些较为常用的改进措施，希望对读者有所帮助。

## 9.1 oVirt 虚拟化平台配置介绍

在开始存储结构优化的实验之前，我们先来了解一下云平台的一般存储配置，包括实例硬盘的创建方式与所在的存储位置，以便基于此进行优化。

### 9.1.1 存储配置背景知识

#### 1. 硬盘分配方式

比较成熟的开源虚拟化或云计算平台包括 OpenStack、CloudStack 和 oVirt 等，它们都有比较清晰的模板和实例概念，同时也有模板与实例镜像存储位置相关的存储域配置。为了区分虚拟机中的模板与实例，本章默认模板即原始虚拟机，所有新实例的创建都依赖于它；实例即依赖模板创建的虚拟机，是具有运行生命周期的虚拟机的通称。

在这些平台中，一般有两种新建实例的方式：克隆与增量。

(1) 克隆，即完整克隆模板配置与硬盘，硬盘的创建方式一般为 cp 或 qemu-img convert。通过上述方式创建的实例，其硬盘与模板硬盘相对独立，在服务器存储上拥有各自的扇区位置，所以在读写操作时受机械硬盘磁头引起的小区域并发问题影响较小，缺点是创建时需要消耗一定的时间，不能满足秒级创建的需求。

(2) 增量，即新建的虚拟机只复制模板配置信息，但其硬盘文件是通过 qemu-img 的 backing file 方法创建的，基于 backing file 创建的硬盘不能是 raw 格式，命令如下：

```
[root@localhost ~]# qemu-img create -b hda.qcow2 -f qcow2 hda-new.qcow2
```

如此创建的硬盘对模板硬盘的依赖较大，非增量文件的读操作（比如启动系统）绝大部分在模板硬盘上进行，所以在传统机械硬盘上多个实例的并发启动风暴更容易发生在此种格式的硬盘上。

#### 2. 集群调度策略

实例的启动在虚拟化平台中的一般调度如图 9-1 所示。

从图 9-1 中我们可以看到，一个实例在真正启动之前会根据一定的策略被安排到某集群的某个主机上，而且根据集群状态可以动态迁移至其他主机，限于篇幅，本章在此将不展开迁移的情况。

我们在实现调度策略的时候应主要考量如下几点：

□ 集群的 CPU 类型 (Family)、集群负载状态。

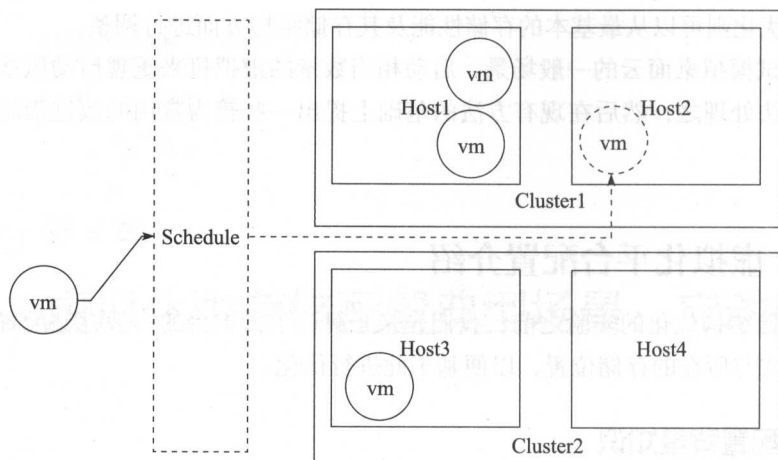


图 9-1 虚拟机调度简图

- ❑ 主机中运行的虚拟机数量、主机历史健康度。
- ❑ 主机 CPU 用度、网络用度、内存用度、存储 I/O 用度，以及它们超过一定阈值的时间。

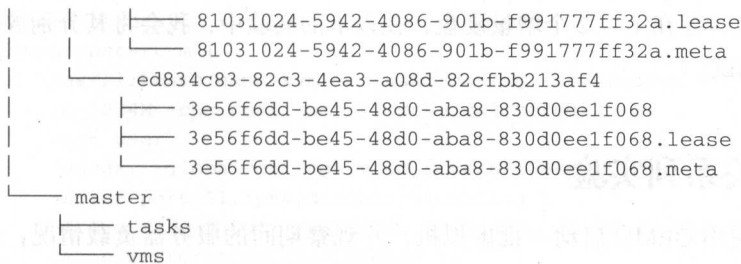
### 9.1.2 模板与实例同一存储

当模板与实例存储在同一文件系统中时，我们创建虚拟机的常用方法有两种——使用模板硬盘的 `hardlink` 作为实例硬盘的 `backing file` 来创建增量硬盘，或者直接复制模板硬盘。

在 oVirt 的数据存储域中我们可以看到如下代码段中的目录结构，其中每个 UUID 都代表了一个对象，`lease` 文件、`metadata` 分别为文件锁和元数据：

```
[root@localhost ~]#tree data/
4440c3f9-balf-4803-86ac-d59c694c61c1/           # 存储域
├── dom_md
│   ├── ids
│   ├── inbox
│   ├── leases
│   ├── metadata
│   └── outbox
├── images
│   ├── d25c4d22-15c1-4f3c-8aef-2ed74deb4723      # 模板硬盘 UUID
│   │   ├── 2bf1ba4c-3fb5-4e25-9368-cc96724b1a63  # 模板硬盘文件
│   │   ├── 2bf1ba4c-3fb5-4e25-9368-cc96724b1a63.lease
│   │   └── 2bf1ba4c-3fb5-4e25-9368-cc96724b1a63.meta
│   ├── d6d7e74d-76fc-48c6-b3c6-3c506b98e73e      # 实例硬盘 UUID
│   │   ├── 2bf1ba4c-3fb5-4e25-9368-cc96724b1a63  # 模板硬盘文件硬链接
│   │   ├── 2bf1ba4c-3fb5-4e25-9368-cc96724b1a63.lease
│   │   ├── 2bf1ba4c-3fb5-4e25-9368-cc96724b1a63.meta
│   └── 81031024-5942-4086-901b-f991777ff32a      # 实例增量硬盘文件
```





模板硬盘和实例硬盘处于同一文件系统上，oVirt 使用了 **hardlink** 链接了模板 `d25c4d22-15c1-4f3c-8aef-2ed74deb4723` 下的硬盘 `2bf1ba4c-3fb5-4e25-9368-cc96724b1a63` 和实例 `6d7e74d-76fc-48c6-b3c6-3c506b98e73e` 下的 `2bf1ba4c-3fb5-4e25-9368-cc96724b1a63`。



**注意** 在同一文件系统上使用 **hardlink** 而非 **softlink** 的原因如下。

- ☐ 节省 **inode** 使用量。
- ☐ 由于省去了访问 **softlink** 的 **inode** 这一步，直接访问源文件的 **inode** 会带来性能上的稍许提升。
- ☐ 改变源文件路径不会影响 **hardlink** 文件的访问。
- ☐ 冗余性及安全性考虑，只有删除了 **inode** 上的所有引用，此 **inode** 才会被删除。

### 9.1.3 模板与实例分离存储

当模板硬盘与实例硬盘分别存储在不同的文件系统中时，就不能使用 **hardlink** 创建虚拟机。oVirt 跨存储域创建实例的方式为克隆创建，而不是直接使用 **backing file** 创建增量硬盘。我们从其存储域的概念就可以推测出它这么做的原因，即：

- ☐ 存储域间没有相关联的模板硬盘与实例硬盘，因此易于存储域的管理（删除、导入存储域与其中的虚拟机）。
- ☐ 对于跨文件系统的存储域，使用复制而不是增量创建更易于减少模板所在存储域的负担。

有时我们在真实部署虚拟桌面的场景中，往往需要多个本地存储域（比如 SAS 与 SSD）混合使用。所以在以下的测试中，我也会使用跨存储域创建增量硬盘的方式。

### 9.1.4 无状态实例的硬盘与快照分离存储

oVirt 中存在一种“无状态”实例，该实例的创建过程如下：

已有模板硬盘 A，我们采用克隆或增量的方式创建实例硬盘 B，勾选“无状态”以后，虚拟机运行会自动创建硬盘 B 的增量硬盘 C，对实例所有的改动都在 C 上，当虚拟机关机之后，平台删除 C。这样一来，所有文件的改变便随之删除，我们称这种工作方式的实例为“无状态”实例。



这种状态下的虚拟机，存在 1 ~ 2 个增量硬盘，在以下的实验中，我会将其分别放置在不同的文件系统中进行测试。

## 9.2 启动风暴相关系列实验

接下来，我们直接使用 QEMU 启动一批虚拟机，并观察期间的服务器负载情况，然后在更换存储配置的情况下再一次启动它们，最后对比两次的服务器负载情况。实验过程中，不考虑 qcow2 格式与 raw 格式的影响，统一使用 qcow2 格式。所有的虚拟机均使用 VirtIO 接口、qcow2 格式硬盘、Windows XP 32 位操作系统，无任何附加软件。同时，为减少 Windows XP 系统启动后对快照硬盘产生额外的硬盘写入操作，比如日志记录、例行任务等系统操作，我在其开机运行 1 个小时之后再行进行模板制作。

实验服务器配置为双路 X5670@2.93GHz、64GB 内存、一块 Intel 480GB 企业级 SSD、一块 WD 1TB 企业级机械硬盘、操作系统为 CentOS 7.1。

### 9.2.1 模板配置

模板的具体配置方法如下：

```
# 虚拟机配置单路单核 1 GB 内存，显示为 Spice 协议
[root@localhost ~]# cat base_xp.sh
#!/bin/bash
/usr/libexec/qemu-kvm -no-user-config -nodefaults \
-m 1024M -cpu host -smp 1,sockets=1,cores=1 \
-net user \
-monitor stdio -vga qxl -global qxl-vga.vram_size=67108864 \
-spice port=7001,ipv4,disable-ticketing \
-drive file=hda.qcow2,if=none,id=drive-virtio-disk0,format=qcow2,cache=none,werror=stop,rerror=stop,aio=threads \
-device virtio-blk-pci,scsi=off,bus=pci.0,addr=0x7,drive=drive-virtio-disk0,id=virtio-disk0,bootindex=1 \
-device virtio-balloon-pci,id=balloon0,bus=pci.0,addr=0x8
```

### 9.2.2 实验脚本

(1) 创建 20 个以 hda.qcow2 为 backing file 的硬盘：

```
[root@localhost ~]# create-imgs.sh
#!/bin/bash
for i in `seq 11 30`
do
qemu-img create -f qcow2 -b hda.qcow2 hda-${i}.qcow2
done
```

(2) 一次性启动 20 台实例：

```
[root@localhost ~]# start-vms.sh
```

```
#!/bin/bash
function startvm {
    /usr/libexec/qemu-kvm -no-user-config -nodefaults \
    -m 1024M -cpu host -smp 1,sockets=1,cores=1 \
    -net user \
    -vga qxl -global qxl-vga.vram_size=67108864 \
    -spice port=$1,ipv4,disable-ticketing \
    -drive file=$2,if=none,id=drive-virtio-disk0,format=qcow2,cache=none,werror=stop,rerror=stop,aio=threads \
    -device virtio-blk-pci,scsi=off,bus=pci.0,addr=0x7,drive=drive-virtio-disk0,id=virtio-disk0,bootindex=1 \
    -device virtio-balloon-pci,id=balloon0,bus=pci.0,addr=0x8
}

for i in `seq 11 30`
do
startvm 70$i hda-$i.qcow2
done
```

### (3) 测量，采样频率为 1Hz:

```
[root@localhost ~]# iostat -cdmx 1|tee 20-xp.iostat-cdm.out
```

(4) 数据预处理，我们只需要总读写速度 (MB/s)、总读写请求 (Requests/s)、CPU 利用率 (%user,%sys) 即可:

```
[root@localhost ~]# awk 'BEGIN {print "cpu usage\n";i=0};$1 ~ /[0-9]/ {print i,$1+$3;i+=1;}' 20-xp.iostat-cdm.out > 20-xp.iostat-cdm-cpu.out
[root@localhost ~]# awk 'BEGIN {print "sdb info\nTime IOPS RMBpsWMBps";i=0};$1 ~ /^sda/ {iops=$4+$5;print i,iops,$6,$7;i+=1;}' 20-xp.iostat-cdm.out > 20-xp.iostat-cdm-sdb.out
[root@localhost ~]# awk 'BEGIN {print "sdb info\nTime IOPS RMBpsWMBps";i=0};$1 ~ /^sdb/ {iops=$4+$5;print i,iops,$6,$7;i+=1;}' 20-xp.iostat-cdm.out > 20-xp.iostat-cdm-sdb.out
```

(5) 可视化示例，由于请求数比速度高很多倍，因此为了方便数据显示我们将请求数除以 10:

```
[root@localhost ~]# cat plot_storage.py
#!/usr/bin/env python
import numpy as np
import matplotlib.pyplot as plt

f_c = file('20-xp.iostat-cdm-sda.out').readlines()
c = np.array(map(str.split,f_c[2:]),dtype='float')

plt.plot(c[:,0],c[:,1]/10,label="IO Requests/s", color="red", linewidth=2)
plt.plot(c[:,0],c[:,2],label="Read MB/s", color="blue", linewidth=2)
plt.plot(c[:,0],c[:,3],label="Write MB/s", color="blue", linewidth=2)
plt.legend()
plt.show()
```

### 9.2.3 WD 1TB 机械硬盘启动 Windows XP 实验

在 WD 1TB 机械硬盘上虚拟机在实例启动期间其服务器的 CPU、I/O 用度分别如图 9-2 和图 9-3 所示，可以看到，系统在第 5 秒左右完成启动，第 8 秒左右自动登录进入桌面，期间 CPU 用度较为平稳、读请求较多。

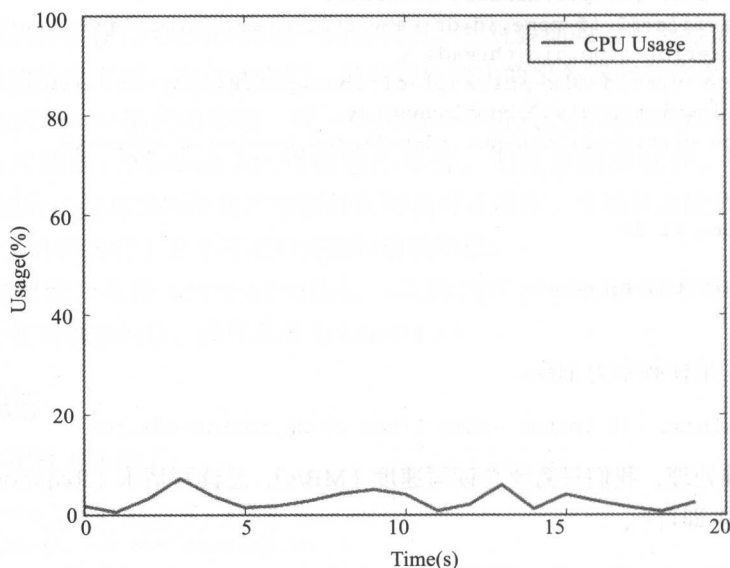


图 9-2 WD 1TB 硬盘上启动单台 XP 时的 CPU 用度

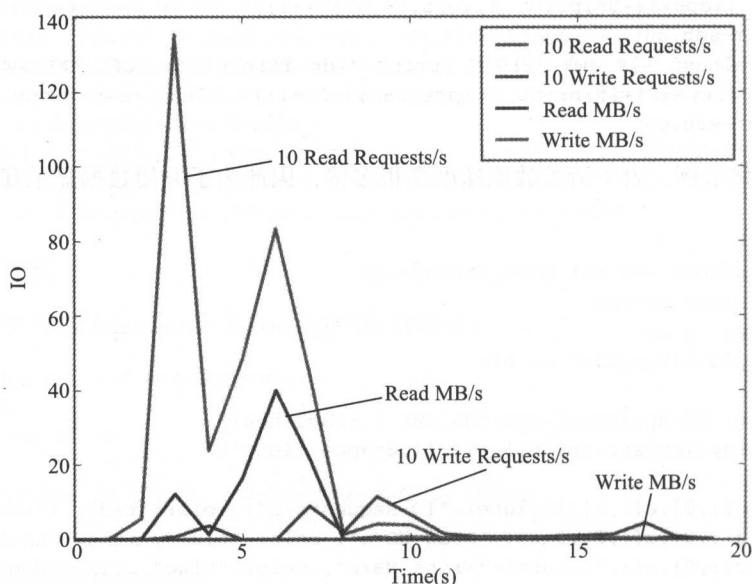


图 9-3 WD 1TB 硬盘上启动单台 XP 时的 I/O 用度

在 WD 1TB 硬盘上 20 台使用增量硬盘的虚拟机实例，在启动期间其服务器的 CPU、I/O 用度分别如图 9-4 和图 9-5 所示。期间虚拟机实例先后进入桌面，在第 300 秒左右全部进入，服务器 CPU 用度在刚开始很繁忙，读请求也很多并且几乎达到硬盘的极限 IOPS，一直持续到登录前，然后开始下降并趋于平稳。

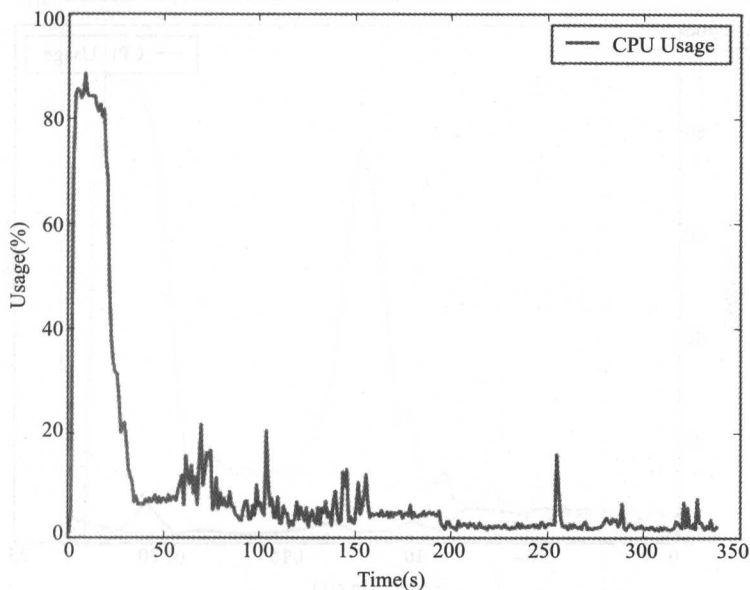


图 9-4 WD 1TB 上启动 20 台 XP 时的 CPU 用度

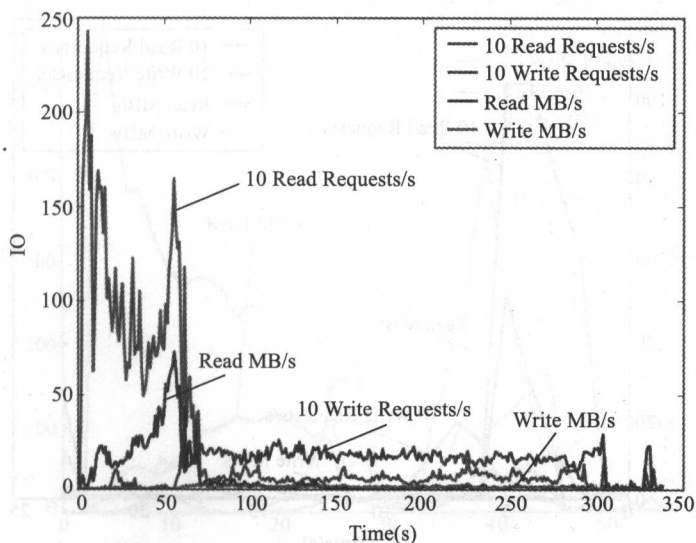


图 9-5 WD 1TB 上启动 20 台 XP 时的 I/O 用度

9.2.4 Intel 480GB SSD 启动 Windows XP 实验

在 Intel 480GB SSD 上使用增量硬盘的虚拟机实例，在启动期间其服务器的 CPU、I/O 用度分别如图 9-6 和图 9-7 所示。虚拟机实例在第 6 秒左右便进入桌面，且服务器 CPU 用度平稳，读写请求数也比较正常。

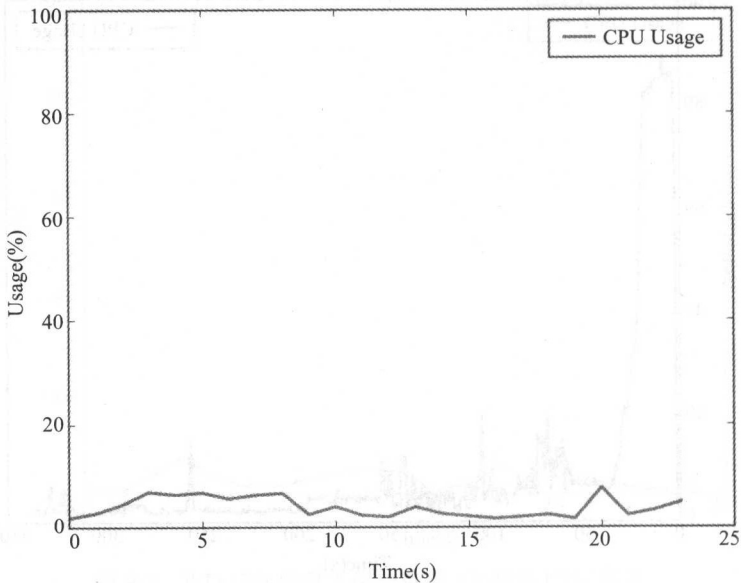


图 9-6 SSD 上启动单台 XP 时的 CPU 用度

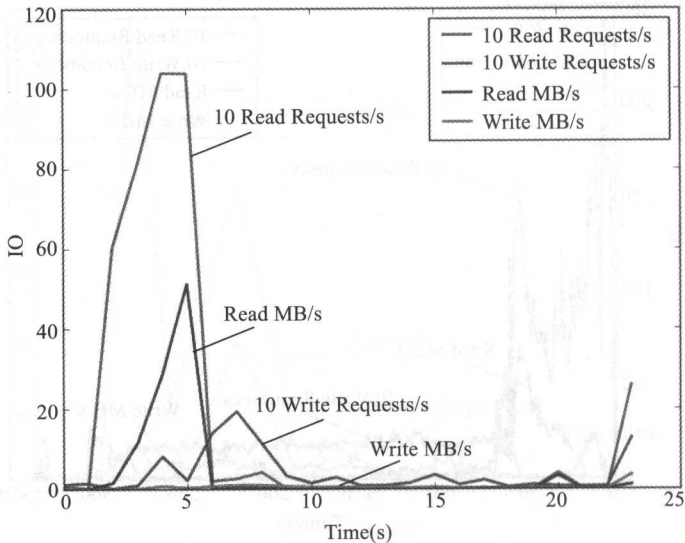


图 9-7 SSD 上启动单台 XP 时的 I/O 用度

在 Intel 480G SSD 上使用增量硬盘的虚拟机实例，在启动期间其服务器的 CPU、I/O 用度分别如图 9-8 和图 9-9 所示。相比于增量硬盘在机械硬盘上时，虚拟机实例在第 35 秒左右便全部进入桌面，期间服务器读请求与预期一致。需要注意的是，由于所有虚拟机在 30 秒左右几乎同时开始登录，所有写请求累加才导致图 9-9 中 I/O 读请求在第 40 秒产生的高峰。

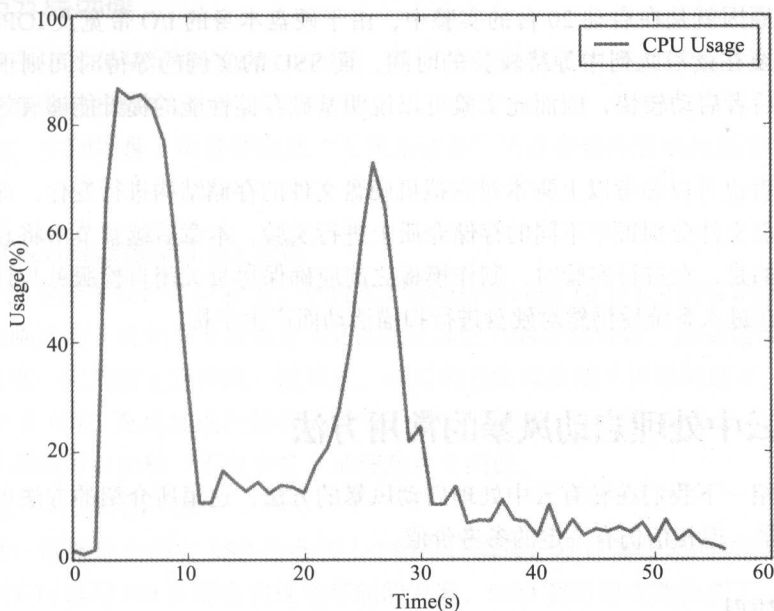


图 9-8 SSD 上启动 20 台 XP 时的 CPU 用度

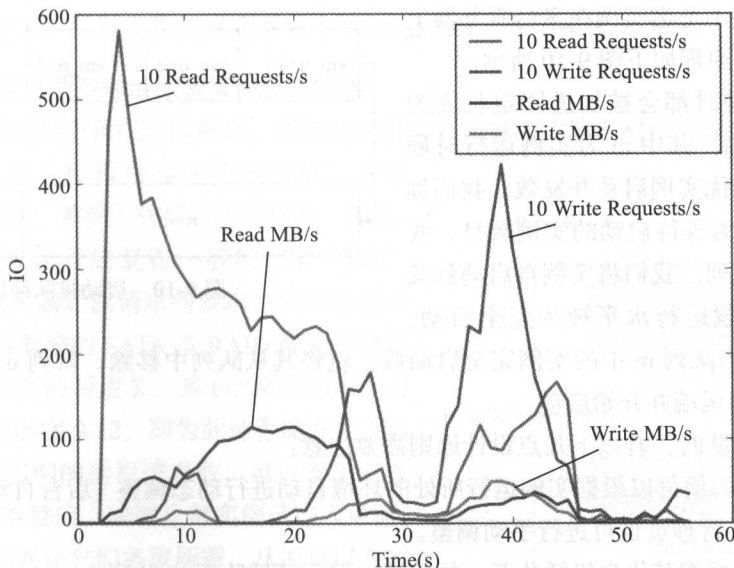


图 9-9 SSD 上启动 20 台 XP 时的 I/O 用度

### 9.2.5 实验结论

从 9.2.3 和 9.2.4 两小节的实验结果可以看出，在同样的硬盘配置下，启动时写请求数量明显高于读请求数，且服务器 CPU 较为繁忙；而使用 SSD 作为存储的虚拟机实例即使是在多个同时启动的情况下，也能较快地将其全部启动，而在机械硬盘存储的情况下启动则比较吃力。原因就是启动 20 台的实验中，由于硬盘本身的 I/O 带宽及 IOPS 限制，WD 1TB 的实例需要在读写队列中等待较长的时间，而 SSD 的实例的等待时间则很短，所以前者启动较慢、后者启动较快，因而此实验可以说明基础存储性能的提升能够有效地改善启动风暴。

同时，读者也可以参考以上脚本对虚拟机硬盘文件的存储结构进行变化，即将增量硬盘文件与模板硬盘文件分别置于不同的存储介质上进行实验，本章后续章节中将直接阐述其结果。需要注意的是，在进行实验时，制作模板之前应确保尽量关闭自检服务与计划任务，防止虚拟机实例在进入系统后仍然对硬盘进行扫描活动而产生干扰。

## 9.3 私有云中处理启动风暴的常用方法

本节将介绍一下我们在私有云中处理启动风暴的方法，这里所介绍的方法可能并不适用于所有应用场景，但相信仍有一定的参考价值。

### 9.3.1 启动排队

启动排队是改善启动风暴比较常见的一种做法，它在虚拟化平台分配实例到某台服务器后执行，即我们不考虑实例在多台服务器上的调度，其基本原理如下图 9-10 所示。

实例在启动时都会被加进固定长度为  $m+n$  的队列末端，其中  $m$  为实例运行时硬盘所在存储的最优实例启动并发数，我们称为启动队列。 $n$  为等待启动的实例数目，我们称为待启动队列。我们将实例自启动到其 IOPS 降至其空载运行水平视为完全启动，此段时间为  $t$ 。当队列  $m$  中的实例完全启动后，就将其从队列中移除，队列  $n$  中的首端的实例加入至  $m$  队列尾端并开始启动。

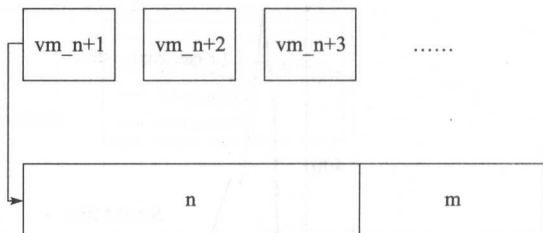


图 9-10 启动排队简图

使用启动排队时，有以下几点设计原则需要注意：

- (1)  $t$  和  $m$  的值可以根据实例运行所处的环境自动进行动态调整（后台自动测量服务器、存储服务水平），管理员也可进行手动调整。
- (2) 实例会根据其优先级插队至  $n$  的首端，但不可插队至  $m$  队列中。



(3) 当队列  $n$  满时, 可以选择拒绝启动或等待资源以启动实例; 某些设计中也可将  $m$  与  $n$  合并为一个启动队列。

此种方式能够比较高效地利用服务器和存储资源, 而且对已启动实例的附带影响又较小。

### 9.3.2 存储分层选择

启动排队的方式是从策略上来解决问题, 接下来我们从实例硬盘存储位置的选择上进一步降低启动风暴带来的影响。

模板硬盘、实例硬盘、增量硬盘及“无状态硬盘”所在存储位置的性能将直接影响实例的启动过程。

#### 1. 统一高速存储改善整体 I/O 分布

从 9.2 节的实验中我们可以看出, 单台 XP 实例启动的前 10 秒是 IOPS 消耗最大的时期。在启动风暴来临之前, 我们只要准备好 IOPS 充足且适当的存储设备, 再适量调整我们的 I/O 调度算法即可在一定程度上缓冲第一波风暴, 而后的登录风暴便不再是问题了。

假设现在有 IOPS 充足且适当的存储设备, 我们将模板、实例、快照硬盘全部放置到这台设备上, 从而就可以比较“不负责任”地缓解这个问题。

为什么说“不负责任”呢?

因为虽然一股脑地全部上 SSD 会改善 I/O 情况而减缓启动风暴带来的影响, 但在成本上对比根据实例实际读写 I/O 分配合理规划存储的方案, SSD 就显得有些浪费了。

当然, 不论是后端存储还是前端服务器 OS, 使用响应时间短、读写带宽大的硬盘总是有益的。现在制造 SSD 的技术正在不断提高, 成本、寿命、速度、容量等与机械硬盘相比, 优势也越来越突出, 所以虚拟化桌面领域中 SSD 会代替机械硬盘成为主流。

#### 2. 增量硬盘高速存储池改善实例 I/O 分布

从 9.2 节的实验中我们可以看出, 实例启动时会有大量的读请求和相对较少的写请求, 而我们的实例就是从模板母盘中读取数据, 并将 CED (创建、修改、删除) 数据写入增量盘。那么, 我们不妨将模板硬盘放置在一般的 SSD (MLC、TLC) 存储上用来满足读请求的要求, 将增量硬盘放置在成本相对低廉的 SATA 盘 RAID 阵列存储上来满足平稳且分散的写请求。其 I/O 路径如图 9-11 所示, 与之对应的图 9-12, 即为此种存储方式下启动 20 台虚拟机实例的读写请求数, 可以看出它在第 60 秒时便基本稳定, 说明全部实例已进入桌面。这样一来, 我们就让它们各取所需, 从而可以在成本上升较少的情况下改善启动风暴带来的影响。

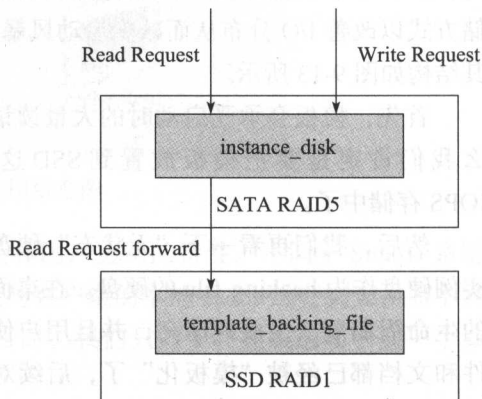


图 9-11 增量硬盘读写请求

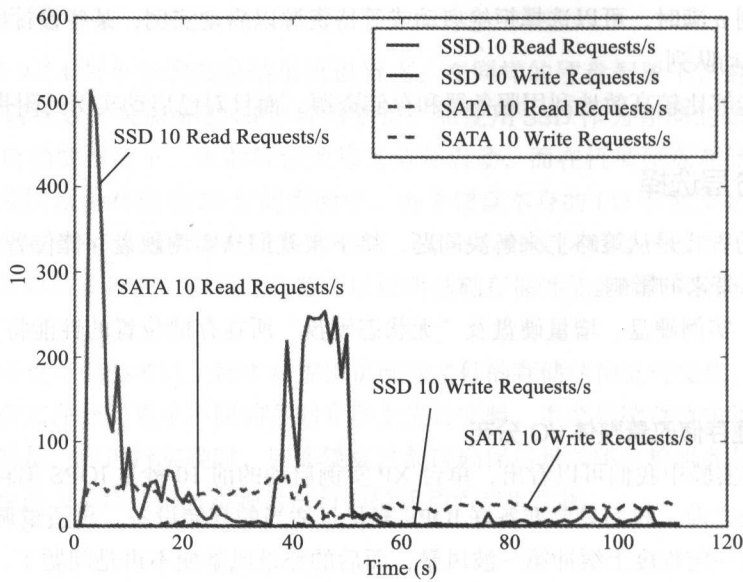


图 9-12 模板位于 SSD，实例位于机械硬盘的读写请求

当然，随着桌面中安装的自启动软件和服务越来越多，而这些后来者都写入了我们的增量盘中，所以增量盘的读请求数在后期会有一定量的上升，从而拖慢整体的读请求水平。根据安装软件的“重度”，我们可以选择提前在模板中安装比较“重”的软件，尽量使增量硬盘中静态数据的（比如文档、媒体文件等）占比扩大，这样就可以使后期的读请求时间较为分散，从而保证系统的启动速度不会受到太多影响。

3. 无状态实例硬盘高速存储池改善实例 I/O 分布

由于无状态实例在桌面云中拥有很重要的位置，所以本节在此介绍一种改善无状态桌面的存储方式以改善 I/O 分布从而减少启动风暴的影响，其结构如图 9-13 所示。

首先，模板会承受启动时的大量读请求，那么 we 肯定是要把模板放置到 SSD 这样的高 IOPS 存储中了。

然后，我们再看一下“无状态”硬盘，即将实例硬盘作为 backing file 的硬盘。在桌面云中它的生命周期短，生成频率高，并且用户使用的软件和文档都已经被“模板化”了，后续对系统很少会产生大量写的情况。所以，它承受的读写请

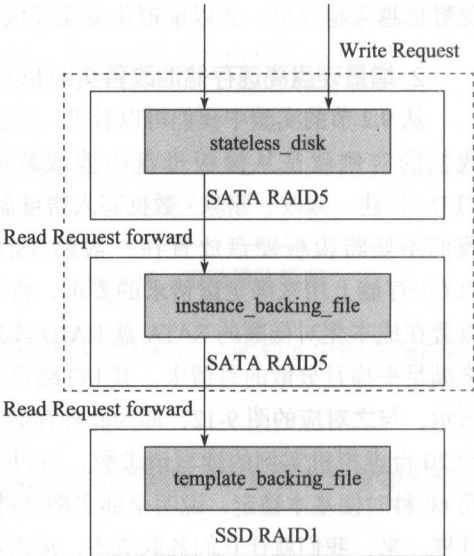


图 9-13 无状态实例的读写请求

求量较低，但是它的每一次生成和删除对所在的物理存储又有频繁的重复写入操作。考虑到 SSD 的擦除次数有限，我们最好将“无状态”硬盘放置在机械硬盘的存储中。

最后，从模板创建的实例硬盘也会承受大量的读请求，但是接下来我们要考虑如下两种状况：当实例硬盘中有较多的拖慢启动时间的软件和服务时，为保证启动时 IOPS 不会成为瓶颈，我们最好把它也放置到 SSD 中；当实例硬盘中静态数据的占比较多时，我们可以优先考虑将其放置在机械盘的阵列存储中，并且可以与“无状态”硬盘放置在同一文件系统中。

### 9.3.3 其他提升桌面云存储性能的方式

#### 1. FS-Cache

FS-Cache 是一种将通过网络获取的数据缓存到本地常驻存储中以加速本地应用的访问，从而减少网络流量的技术，其使用过程如图 9-14 所示。

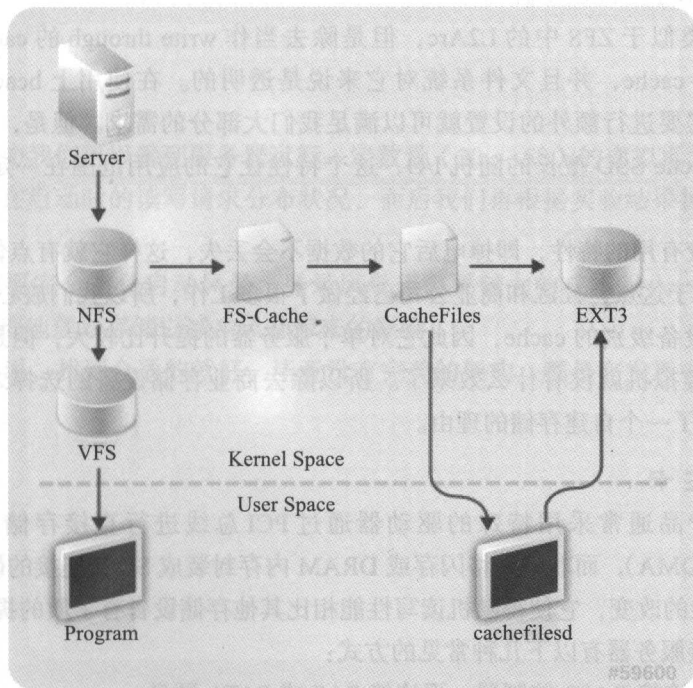


图 9-14 FS-Cache 使用示意图

FS-Cache 在设计之初，就尽量保持其对管理员和用户透明的特性。不同于 Solaris 系统的 cacheefs，FS-Cache 允许服务器端的文件系统直接与客户端的本地 cache 进行交互，而不需要额外挂载文件系统。比如在 NFS 上使用 FS-Cache 时，我们只需要在挂载选项中加一个参数就可以启用它了。

FS-Cache 在不改变网络文件系统基本操作的前提下，在文件系统中提供了一个常驻

cache 用于数据的暂存。比如，一个 NFS 客户端即使之前配置了 FS-Cache 并且有一部分数据已被 cache，关闭它后仍然可以挂载 NFS 并且使用被 cache 的数据部分。FS-Cache 也可隐藏掉客户端文件系统驱动的所有 I/O 错误。在配置 FS-Cache 的时候，我们需要一个 cache 的后端，这个后端的文件系统需要支持 bmap 和扩展属性。

但 FS-Cache 不能 cache 任意网络文件系统，它需要的共享文件系统驱动要能够与 FS-Cache 交互、存储数据、建立和验证元数据。FS-Cache 通过 cache 后端文件系统中数据的键索引和一致性检查来保证数据的持久特性，即数据的有效性校验。

FS-Cache 在私有云中一般适用于教学、办公等模板相同的批量桌面，不适用于存储模板大多相异的虚拟服务器。

## 2. bcache

bcache 是 Linux 内核块设备层的 cache 模块，它可以将一块或多块 SSD 用作为普通硬盘的 cache，有点类似于“混合硬盘”。

bcache 有点类似于 ZFS 中的 L2Arc，但是除去当作 write through 的 cache 外，它也可当作 write back 的 cache，并且文件系统对它来说是透明的。在使用上 bcache 可以很方便地启用，并且不需要进行额外的设置就可以满足我们大部分的需求。但是，它不会去 cache 顺序 I/O，只会 cache SSD 擅长的随机 I/O，这个特性让它的应用范围在一定程度上有所限制了。

还有一点比较有用的特性，即掉电后它的数据不会丢失，这样它就有点像一个带电池的 raid 控制器了。关于这点，社区和商业公司已经做了很多工作，所以我们放心用就好了。

由于它是块设备级别的 cache，因此它对单个服务器的提升比较大，但是对于运行在外接存储设备上的虚拟机就没有什么效果了。所以除去商业存储设备的选择之外，bcache 和 FScache 让我们多了一个自建存储的理由。

## 3. SSD PCI-E 卡

PCI-E SSD 产品通常采用特殊的驱动器通过 PCI 总线进行直接存储器访问 (Direct Memory Access, DMA)，而非只是将闪存或 DRAM 内存封装成 SCSI 连接的硬盘驱动器。这是一种比较革命性的改变，它使得随机读写性能相比其他存储设备有了质的提升。

存储设备连接服务器有以下几种常见的方式：

- ❑ PCI-E 总线连接 RAID 控制器，再连接 SAS 或 SATA 硬盘。
- ❑ 通过 PCI-E 总线连接 HBA 卡，再连接到硬盘阵列。
- ❑ SSD 挂载到 PCI-E SSD 卡，再挂载到 PCI-E 插槽。

我们可以看到 CPU 通过 PCI-E SSD 卡提供的短路径来访问 SSD，结合 flash 的高速读写性能，极大地提升了存储性能，突破了存储的 I/O 瓶颈。

但因为其连接 SSD 的数目有限，且单卡成本较为高昂，所以对于旨在降低用户成本、提高管理效率的私有云桌面来说，目前这是一种相对“奢侈”的解决方案。

### 拓展：其他存储 I/O 风暴介绍

私有云中有几种 I/O 风暴比较常见，比如登录风暴、杀毒风暴。登录风暴发生在系统启动后但还未进入桌面时，多人同时登录而对服务器造成压力的情况，这点我们从图 9-9 和图 9-12 中可以明显看出。对于登录风暴，我们可以使用自动登录的方式，将其与启动划分到同一顺序过程中，这样启动排队策略也能作用于它了。

而杀毒风暴则是指多个虚拟机实例的某些应用程序在对硬盘分区乃至全盘同时进行扫描时，对服务器存储产生的大量压力而导致系统响应缓慢的现象，这些应用程序包括杀毒软件、监控软件等。开源私有云平台中的杀毒风暴可以利用 GuestFS 工具作为解决方案，即将虚拟机的硬盘挂载到统一的杀毒服务器上，从而可以在实例离线状态下进行查杀、在离线状态下进行扫描。

## 9.4 小结

从模拟试验中我们可以看到服务器运行一定数量（20 ~ 50）的虚拟桌面下的存储 I/O 负载，了解到桌面在启动时的读写请求分布状况，而后我们再根据实验结果提出桌面功能的简单优化策略。

总之，目前要完全避免启动风暴、登录风暴还需要较大的成本投入，而在私有桌面云中，我们最好合理地规划存储以减少风暴带来的影响。

对于读者来说，找到合适的就好，毕竟没有完美的架构，都是在发展中逐渐进化而满足甚至超越需求的。

## 私有云桌面网络组建

### 作者简介

蒋迪，上海沃帆信息科技有限公司资深虚拟化基础架构工程师，从事云平台研发工作，主要包括 OpenStack、oVirt 的虚拟化、网络、存储组件及其在桌面云领域的应用。涉及虚拟化产品应用（云桌面、云服务器），云平台架构与相关开发（自主产品与开源软件），嵌入式系统与设备（ARM 架构），分布式存储（GlusterFS）。多次参与银行与教育领域的私有云架构设计与实施。

2016 年年初，Amazon 刚刚推出 Workspace，即公有云形式的桌面。在此之前的几年中，桌面云主要是以私有云的形式存在，主要原因除了安全性以外，还有就是桌面协议性能所引起的带宽与延迟问题。虽然带宽问题在私有云中并不是很明显，但是仍然需要合理规划以增强其整体安全性与利用率。本章将介绍私有云桌面中组网的基础技术及其实现方式，然后再列举 oVirt 和 OpenStack 中关于虚拟桌面网络的组建方式。最后我将尝试从多个方面解析典型私有云桌面中网络配置的选择，希望对私有云厂商同仁及企业网管有一定的帮助。

### 10.1 桌面云常用网络

我所接触的虚拟化桌面云平台中，其组网实现可以是 VLAN、NAT、桥接、OpenvSwitch 等。接下来本章将介绍如何在 libvirt/QEMU 组合中使用这些网络。

#### 10.1.1 NAT 网络

对于 NAT（Network Address Translation，网络地址转译）网络，我们或许并不陌生。一



般家庭路由器接入网络提供商（ISP）时，路由器会获得一个公网 IP（或者广域网 IP），局域网电脑接入路由器后获得内网 IP，用户就是通过 NAT 来访问互联网的。

NAT 的原理是在客户端发出的 IP 封包到达 NAT 网关后，其中的源 IP 地址会被替换为网关 IP（这个过程称为 SNAT，源 IP 和端口会被记录）再继续发送至外部服务器，当服务器将 IP 封包返回到 NAT 网关后，其中的目的地址将变更为客户端 IP（这个过程称为 DNAT，它会利用 SNAT 记录的 IP 与端口信息）再传递回客户端，外部地址一般不能直接访问 NAT 后端的地址（需要在 NAT 网关打开端口映射），其基本过程如图 10-1 所示。如果图 10-1 中的请求是从外网客户端发起的，内网服务器做出响应，经过 DNAT 和 SNAT 之后返回，那就是我们常说的端口映射了。

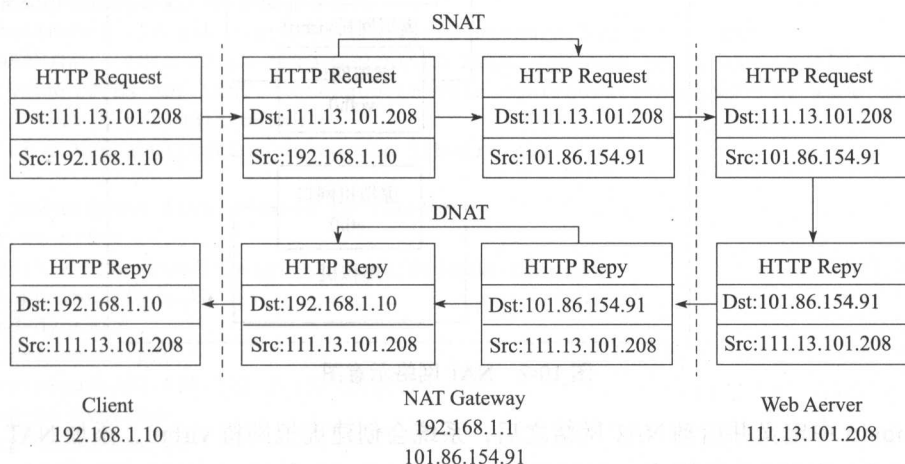


图 10-1 NAT 实现

NAT 在 libvirt/QEMU 组合中的默认结构如图 10-2 所示，对应到路由器中，物理网口 eth0 就相当于路由器的 WAN 接口，虚拟网口 veth0 就相当于路由器的 LAN 接口，虚拟机对应着我们的客户端，NAT 即在主机网络协议栈中进行包的地址转译。

为了在 libvirt 中使用 NAT 网络，我们首先需要在 libvirt 中定义一个名为 default 的 NAT 网络的配置：

```
[root@node1 ~]# cat> default-nat.xml<<EOF
<network>
<name>default-nat</name>
<forward mode='nat' />
<bridge name='virbr0' stp='on' delay='0' />
<ip address='192.168.122.1' netmask='255.255.255.0' >
<dhcp>
<range start='192.168.122.2' end='192.168.122.254' />
</dhcp>
</ip>
</network>
```



```
[root@node1 ~]#virsh net-define default-nat.xml
[root@node1 ~]#virsh net-start default-nat
[root@node1 ~]#virsh net-autostart default-nat
```

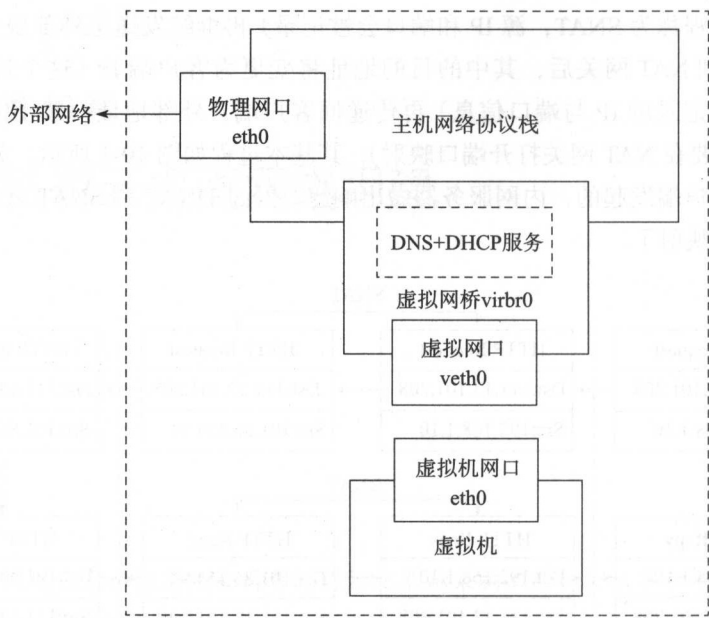


图 10-2 NAT 网络示意图

在 libvirt 中定义并启动 NAT 网络之后，系统会创建虚拟网桥 virbr0、添加 NAT 规则并在虚拟网桥上启动 DHCP 服务。添加的 NAT 规则（SNAT 或 MASQUERADE 皆可，主要区别是在 SNAT 期间前者的内核将一直记录连接信息，网口关闭后连接虽然失效但信息仍然存在，当接口重启后连接会继续生效；而后者的连接信息将随着网口的关闭而丢失，除非虚拟机再次发起连接）会将来自 virbr0 的流量传递至物理网口 eth0，DHCP 服务则由轻量级的 dnsmasq 提供，主机需要做的配置如下所示：

```
# 查看网桥
[root@node1 ~]# brctl show virbr0
bridge name      bridge id        STP enabled    interfaces
virbr0           8000.52540026be99  yes            virbr0-nic

# 查看防火墙 NAT 规则
[root@node1 ~]# iptables -L
Chain INPUT (policy ACCEPT)
target prot opt source                destination
ACCEPT  udp  --  anywhere                anywhereudpdpt:domain
ACCEPT  tcp  --  anywhere                anywheretcpdpt:domain
ACCEPT  udp  --  anywhere                anywhereudpdpt:bootps
ACCEPT  tcp  --  anywhere                anywheretcpdpt:bootps

Chain FORWARD (policy ACCEPT)
```

```

targetprot opt source destination
ACCEPT all -- anywhere 192.168.122.0/24 ctstate
RELATED,ESTABLISHED
ACCEPT all -- 192.168.122.0/24 anywhere
ACCEPT all -- anywhere anywhere
REJECT all -- anywhere anywhere reject-with icmp-
port-unreachable
REJECT all -- anywhere anywhere reject-with icmp-
port-unreachable

```

Chain OUTPUT (policy ACCEPT)

```

targetprot opt source destination
ACCEPT udp -- anywhere anywhereudpdp:bootpc

```

# 查看 dnsmasq 配置, 即 DHCP 服务配置

```
[root@node1 ~]# cat /var/lib/libvirt/dnsmasq/default-nat.conf
```

```

##WARNING: THIS IS AN AUTO-GENERATED FILE. CHANGES TO IT ARE LIKELY TO BE
##OVERWRITTEN AND LOST. Changes to this configuration should be made using:

```

```
## virsh net-edit default
```

```
## or other application using the libvirt API.
```

```
##
```

```
## dnsmasqconf file created by libvirt
```

```
strict-order
```

```
pid-file=/var/run/libvirt/network/default-nat.pid
```

```
except-interface=lo
```

```
bind-dynamic
```

```
interface=virbr0
```

```
dhcp-range=192.168.122.2,192.168.122.254
```

```
dhcp-no-override
```

```
dhcp-lease-max=253
```

```
dhcp-hostsfile=/var/lib/libvirt/dnsmasq/default-nat.hostsfile
```

```
addn-hosts=/var/lib/libvirt/dnsmasq/default-nat.addnhosts
```

然后在虚拟机设备段中添加如下定义, 即可通过 DHCP 获得地址并以 NAT 方式访问外部网络:

```

<interface type='network'>
<source network='default-nat' />
<model type='rtl8139' />
</interface>

```

在虚拟化中使用 NAT 的好处不言而喻, 主要有以下几点:

- 解决了局域网中 IP 不足的隐患。
- 网络拓扑更具弹性。
- 客户端网络行为更易控制, 比如局域网或外网访问权限。

## 10.1.2 桥接网络

桥接是将两个以上的网络通过设备进行连接从而达到网络汇聚的目的。桥接位于 OSI 模

型中的数据链路层。桥接技术共有 4 种：简单桥接、多端口桥接、透明桥接和源路由桥接，具体如表 10-1 所示：

表 10-1 桥接技术

桥接技术	简介
简单桥接	简单桥接即将两个网段通过最简单的方式进行连接，比如网口直连，从而达到网段间相通的目的，其中数据包是通过存储 - 转发的形式进行传递的
多端口桥接	多端口桥接是在简单桥接的基础上，将多个网段中的端口进行桥接。它是网络交换机的技术核心
透明桥接	透明桥接是具有学习功能的多端口桥接，即数据包在传递时会将包中的源 MAC 地址、目的 MAC 地址及对应的端口保存到数据库中，从而避免包在每次传递时都要在全部端口上进行泛洪 (flooding) 定位。现代交换机中的一般实现方式就是透明桥接
源路由桥接	源路由桥接通常应用在令牌环网络中，主机需要知道它所连接的网桥标识，是一种非透明网桥

在虚拟化中，我们通常使用的是透明桥接，它一般是将主机的物理网口与虚拟机网口进行桥接，虚拟机网口的流量虽然是通过物理网口进入外部网络的但地位仍与物理网口相同。桥接是很多云平台都有提供的组网方式，主机与虚拟机之间的连接示意图如图 10-3 所示。

使用桥接网络，我们首先需要创建网桥 br0，然后将物理网口 eno16777736（同图 10-3 中的物理网口 eth0）添加至 br0 中：

```
# 将 eth0 作为 br0 的桥接接口
[root@node1 ~]# cat>/etc/sysconfig/network-scripts/ifcfg-eno16777736 <<EOF
DEVICE=eno16777736
ONBOOT=yes
BRIDGE=br0
NM_CONTROLLED=no
EOF

# 定义网桥 br0
[root@node1 ~]# cat> /etc/sysconfig/network-scripts/ifcfg-br0 <<EOF
DEVICE=br0
TYPE=Bridge
BOOTPROTO=static
IPADDR=192.168.0.140
NETMASK=255.255.255.0
GATEWAY=192.168.0.1
DNS1=192.168.0.1
ONBOOT=yes
DELAY=0
NM_CONTROLLED=no
```

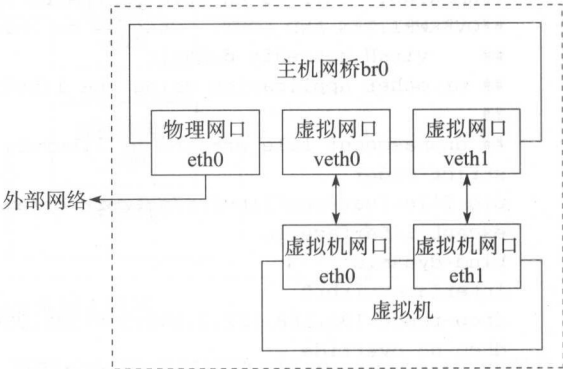


图 10-3 桥接网络示意图

EOF

# 重启网络使设置生效

```
[root@node1 ~]# service network restart
```

# 查看网桥信息

```
[root@node1 ~]# brctl show
```

bridge name	bridge id	STP enabled	interfaces
br0	8000.000c29c93e66	no	eno16777736

在 libvirt 中定义一个桥接网络的配置:

# 在 libvirt 中定义此网络, 名称为 host-bridge

```
[root@node1 ~]# cat>bridge.xml<<EOF
```

```
<network>
```

```
  <name>host-bridge</name>
```

```
  <forward mode='bridge' />
```

```
  <bridge name='br0' />
```

```
</network>
```

EOF

```
[root@node1 ~]# virsh net-define bridge.xml
```

```
Network host-bridge defined from bridge.xml
```

# 启动此网络, 并将其标记为随 libvirt 启动

```
[root@node1 ~]# virsh net-start host-bridge
```

```
Network host-bridge started
```

```
[root@node1 ~]# virsh net-autostart host-bridge
```

```
Network host-bridge marked as autostarted
```

然后在虚拟机的设备段定义中添加如下配置即可:

```
<interface type='network'>
```

```
  <source network='host-bridge' />
```

```
  <model type='rtl8139' />
```

```
</interface>
```

启动虚拟机后, 可以查看 br0 桥接的桥接状况:

```
[root@node1 ~]# brctl show br0
```

bridge name	bridge id	STP enabled	interfaces
br0	8000.000c29c93e66	no	eno16777736

vnet0

其中 br0 上的 vnet0 接口是由 libvirt 自动创建的与虚拟机 eth0 相连的虚拟 TAP 设备接口, 我们也可以通过 ip 命令进行手动创建, 然后在 QEMU 命令中指定使用此接口, 代码如下所示:

# 创建 TAP 设备接口, 也可以使用命令 “tunctl -u root -t tap0”

```
[root@node1 ~]# iptuntnet add tap0 mode tap
```

# 将 tap0 加入 br0 中

```
[root@node1 ~]# brctladdif br0 tap0
```

```
[root@node1 ~]# brctl show br0
```

bridge name	bridge id	STP enabled	interfaces
-------------	-----------	-------------	------------

```
br0                8000.000c29c93e66    no                eno16777736
                                tap0
# 启动虚拟机测试
[root@node1 ~]#qemu-system-x86_64 -net nic -net tap,ifname=tap0,script=no-
hdahda.qcow2
```

### 10.1.3 VLAN 网络

QEMU 虚拟机使用 802.1q VLAN 网络有两种模式，即 Access 模式与 Trunk 模式。使用 Access 模式需要在主机中将指定的 tagged VLAN 接口（比如 eth0.123）加入对应的网桥，虚拟机的数据包将在流入 / 流出网桥时将被添加 / 剥离标签；Trunk 模式下的物理接口（比如 eth0）将直接加入网桥，虚拟机在其网口上主动添加 / 剥离 VLAN 标签后才能与外部通信。某些发行版内核可能不支持 QEMU 虚拟机的 Trunk 模式，一般需要借助 ebttables 工具将二层帧进入 broute 时对其进行路由处理（routing）而不是桥接处理（bridging）（命令形如“ebttables -t broute -A BROUTING -i enp3s0 -p 802\_1Q -j DROP”）。

接下来笔者用 libvirt/QEMU 进行示例，主机的物理网口 enp3s0 与交换机上的 Hybrid 接口相连，它属于 untagged VLAN 1 及 tagged VLAN 10、11。

#### 1. Access 模式

Access 模式在交换机中表示为只允许 untagged 流量进入的网口，在 libvirt 中它表示为与虚拟机网口相连的网桥是 Access 模式。此时我们需要在主机中创建三个网桥 br1、br10、br11，同时物理网口 enp3s0 及其 VLAN 子网口 enp3s0.10、enp3s0.11 分别加入到这三个网桥中，虚线部分的右侧网桥是 untagged 流量，然后流向左侧主机时被添加标签成为 tagged 流量，反之亦然，如图 10-4 所示。

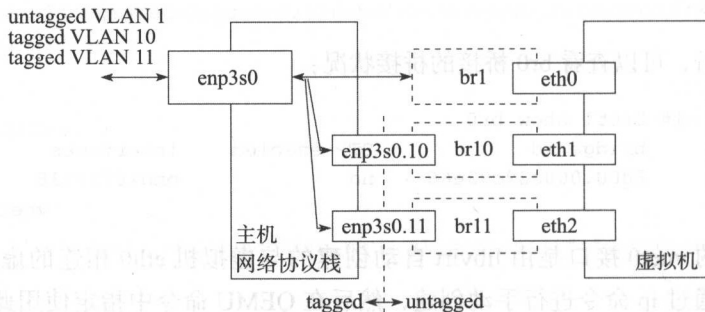


图 10-4 Access 模式虚拟机流量路径

创建三个网桥并将对应的 VLAN 子接口分别加入其中：

```
# 定义 enp3s0 及其 VLAN 子接口 enp3s0.10、enp3s0.11
[root@node1 ~]# cat >/etc/sysconfig/network-scripts/ifcfg-enp3s0 <<EOF
DEVICE=enp3s0
```

```

ONBOOT=yes
BRIDGE=br1
NM_CONTROLLED=no
EOF
[root@node1 ~]# cat>/etc/sysconfig/network-scripts/ifcfg-enp3s0.10 <<EOF
DEVICE=enp3s0.10
ONBOOT=yes
BRIDGE=br110
VLAN=yes
NM_CONTROLLED=no
EOF
[root@node1 ~]# cat>/etc/sysconfig/network-scripts/ifcfg-enp3s0.11 <<EOF
DEVICE=enp3s0.11
ONBOOT=yes
BRIDGE=br11
VLAN=yes
NM_CONTROLLED=no
EOF

# 定义网桥 br1、br10、br11
[root@node1 ~]# cat> /etc/sysconfig/network-scripts/ifcfg-br1<<EOF
DEVICE=br1
TYPE=Bridge
BOOTPROTO=static
IPADDR=192.168.0.141
NETMASK=255.255.255.0
GATEWAY=192.168.0.1
DNS1=192.168.0.1
ONBOOT=yes
DELAY=0
NM_CONTROLLED=no
EOF
[root@node1 ~]# cat> /etc/sysconfig/network-scripts/ifcfg-br10<<EOF
DEVICE=br10
TYPE=Bridge
BOOTPROTO=static
IPADDR=172.20.10.141
NETMASK=255.255.255.0
GATEWAY=172.20.10.254
ONBOOT=yes
DELAY=0
NM_CONTROLLED=no
EOF
[root@node1 ~]# cat> /etc/sysconfig/network-scripts/ifcfg-br11<<EOF
DEVICE=br11
TYPE=Bridge
BOOTPROTO=static
IPADDR=172.20.11.141
NETMASK=255.255.255.0
GATEWAY=172.20.11.254

```

```
ONBOOT=yes
DELAY=0
NM_CONTROLLED=no
EOF
```

然后在 libvirt 中定义 3 个网络：

# 在 libvirt 中定义此网络，名称为 host-bridge

```
[root@node1 ~]# cat> vlan1.xml<<EOF
<network>
```

```
  <name> vlan1</name>
```

```
  <forward mode='bridge' />
```

```
  <bridge name='br1' />
```

```
</network>
```

```
EOF
```

```
[root@node1 ~]# cat> vlan10.xml<<EOF
```

```
<network>
```

```
  <name> vlan10</name>
```

```
  <forward mode='bridge' />
```

```
  <bridge name='br10' />
```

```
</network>
```

```
EOF
```

```
[root@node1 ~]# cat> vlan11.xml<<EOF
```

```
<network>
```

```
  <name> vlan11</name>
```

```
  <forward mode='bridge' />
```

```
  <bridge name='br11' />
```

```
</network>
```

```
EOF
```

```
[root@node1 ~]# virsh net-define vlan1.xml
```

```
Network vlan1 defined from vlan1.xml
```

```
[root@node1 ~]# virsh net-define vlan10.xml
```

```
Network host-bridge defined from vlan10.xml
```

```
[root@node1 ~]# virsh net-define vlan11.xml
```

```
Network host-bridge defined from vlan11.xml
```

# 启动网络，并将其标记为随 libvirt 启动

```
[root@node1 ~]# virsh net-start vlan1
```

```
Network vlan1 started
```

```
[root@node1 ~]# virsh net-start vlan10
```

```
Network vlan10 started
```

```
[root@node1 ~]# virsh net-start vlan11
```

```
Network vlan11 started
```

```
[root@node1 ~]# virsh net-autostartvlan1
```

```
Network vlan1 marked as autostarted
```

```
[root@node1 ~]# virsh net-autostartvlan10
```

```
Network vlan10 marked as autostarted
```

```
[root@node1 ~]# virsh net-autostartvlan11
```

```
Network vlan11 marked as autostarted
```

最后在虚拟机设备段中添加如下定义，启动后在虚拟机系统内设置对应 VLAN 地址段的



网络信息（不需要添加 VLAN 标签）即可接入网络：

```
<interface type='network'>
<source network='vlan1' />
<model type='rtl8139' />
</interface>
<interface type='network'>
<source network='vlan10' />
<model type='rtl8139' />
</interface>
<interface type='network'>
<source network='vlan11' />
<model type='rtl8139' />
</interface>
</interface>
```

## 2. Trunk 模式

Trunk 模式在交换机中表示为只允许 tagged 流量进入的网口，在 libvirt 中它表示为与虚拟机网口相连的网桥是 Trunk 模式。此时我们只需要在主机中创建一个网桥 br0，将物理网口 enp3s0 加入网桥，但需要同时设置 br0 的 VLAN 子接口。虚拟机的三个网口与网桥 br0 相连，虚线部分的左侧网桥中是 tagged 流量，然后流向右侧虚拟机时标签被剥离成为 untagged 流量，如图 10-5 所示。

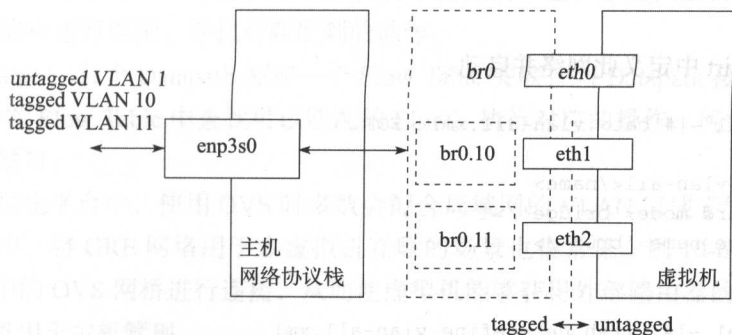


图 10-5 Access 模式虚拟机流量路径

创建一个网桥 br0，并将物理网口 enp3s0 加入其中：

```
[root@node1 ~]# cat>/etc/sysconfig/network-scripts/ifcfg-enp3s0 <<EOF
DEVICE=enp3s0
ONBOOT=yes
BRIDGE=br0
NM_CONTROLLED=no
EOF
[root@node1 ~]# cat> /etc/sysconfig/network-scripts/ifcfg-br0<<EOF
DEVICE=br0
```

```

TYPE=Bridge
BOOTPROTO=static
IPADDR=192.168.0.141
NETMASK=255.255.255.0
GATEWAY=192.168.0.1
DNS1=192.168.0.1
ONBOOT=yes
DELAY=0
NM_CONTROLLED=no
EOF

```

创建 br0 的两个 VLAN 子接口，由于 VLAN 子接口是定义到网桥上的，所以当流量到达网桥时 VLAN 标签会被保持，代码如下所示：

```

[root@node1 ~]# cat>/etc/sysconfig/network-scripts/ifcfg-br0.10 <<EOF
DEVICE=br0.10
ONBOOT=yes
VLAN=yes
NM_CONTROLLED=no
EOF
[root@node1 ~]# cat>/etc/sysconfig/network-scripts/ifcfg-br0.11 <<EOF
DEVICE=br0.11
ONBOOT=yes
BRIDGE=br0
VLAN=yes
NM_CONTROLLED=no
EOF

```

然后在 libvirt 中定义此网络并启动：

```

[root@node1 ~]# cat> vlan-all.xml<<EOF
<network>
  <name>vlan-all</name>
  <forward mode='bridge'/>
  <bridge name='br0' />
</network>
EOF
[root@node1 ~]# virsh net-define vlan-all.xml
Network vlan-all defined from vlan-all.xml
# 启动网络，并将其标记为随 libvirt 启动
[root@node1 ~]# virsh net-start vlan-all
Network vlan-all started
[root@node1 ~]# virsh net-autostartvlan-all
Network vlan-all marked as autostarted

```

最后在虚拟机设备段中添加如下定义，启动后在虚拟机系统内设置网口 eth0 及要加入的 VLAN 子标签（比如 eth0.10、eth0.11）的网络信息即可接入网络：

```

<interface type='network'>
<source network='vlan-all'/>

```

```
<model type='rtl8139' />
</interface>
```

### 10.1.4 Open vSwitch

Open vSwitch (下面简称为 OVS) 是由 Nicira Networks 主导的, 运行在虚拟化平台上的虚拟交换机。OVS 可以为动态变化的端点提供两层交换功能, 从而很好地控制虚拟网络中的访问策略、网络隔离、流量监控等。OVS 遵循 Apache 2.0 许可证, 能同时支持多种标准的管理接口和协议。OVS 也提供了对 OpenFlow 协议的支持, 用户可以使用与 OpenFlow 协议兼容的任意控制器对 OVS 进行远程管理控制, 如果配合 SDN 交换机则可较大规模地替代传统网管方式。

在 OVS 中, 有几个非常重要的概念。

- ❑ Bridge: Bridge 代表一个虚拟交换机 (vSwitch), 一个主机中可以创建一个或多个 Bridge 设备。
- ❑ Port: 端口与物理交换机的端口概念类似, 每个 Port 都隶属于一个 Bridge。
- ❑ Interface: 连接到 Port 的网络接口设备。在通常情况下, Port 和 Interface 是一对一的关系, 只有在将 Port 配置为 bond 模式后, Port 和 Interface 才是一对多的关系。
- ❑ Controller: OpenFlow 控制器。OVS 可以同时接受一个或多个 OpenFlow 控制器的管理。
- ❑ Datapath: 在 OVS 中, Datapath 负责执行数据交换, 也就是把从接收端口收到的数据包在流表中进行匹配, 并执行匹配到的动作。
- ❑ Flow Table: 每个 Datapath 都和一个 Flow Table 关联, 当 Datapath 接收到数据之后, OVS 会在 Flow Table 中查找可以匹配的 Flow, 执行对应的操作, 例如转发数据到其他指定端口。

一般在虚拟化平台中, 使用 OVS 时多数会配合局域网的 VLAN 以便于管理。在流量较小的网络环境中, 将 GRE 网络用于多虚拟机互联的场景也很常见。图 10-6 是使用 GRE 隧道将两个主机中的 OVS 网桥进行连接, 从而使虚拟机能够获得外部路由器的地址进行上网, 其中内部交换机用于主机管理。

在 libvirt 中使用 OVS 网络时, 仅需要进行如下定义:

```
# cat ovs-br0.xml
<network>
<name>ovs-net</name>
<forward mode='bridge' />
<bridge name='ovs-br0' />
<virtualport type='openvswitch' />
</network>

# virsh define ovs-br0.xml
# virsh start ovs-br0
```

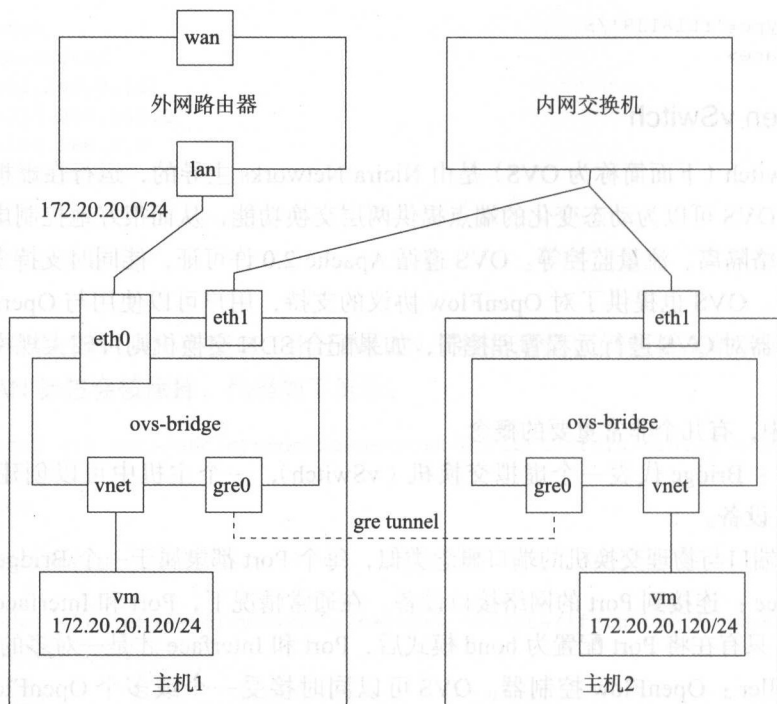


图 10-6 OVS 使用 GRE 管道网络示意图

## 10.2 oVirt/OpenStack 的桌面网络应用

oVirt 的虚拟机网络默认使用桥接，而 OpenStack 可以选配使用桥接、OVS 等作为后端。本节将以 oVirt 为主，OpenStack 网络为辅，为大家介绍一些典型的桌面云网络配置场景。

### 10.2.1 oVirt/OpenStack 组网方式

oVirt/OpenStack 中的组网方式都以 10.1 节所介绍的三种组网为基础，搭配 VLAN (tagged、untagged) 进行组合。

#### 1. OpenStack 典型组网方式

OpenStack 的组网后端有 Linux Bridge 和 OpenvSwitch 两种，在绝大多数的实施中我们一般都会使用 VLAN 配合组网。

##### (1) Linux Bridge 组网

使用 Linux Bridge 作为 Neutron 后端时，网络节点会使用诸如 ip、dnsmasq、iptables、brctl 等命令完成二三层功能（Linux Bridge 网络节点总体情况如图 10-7 所示）。每一个接口与桥的信息都会以元数据的形式保留在数据库中。

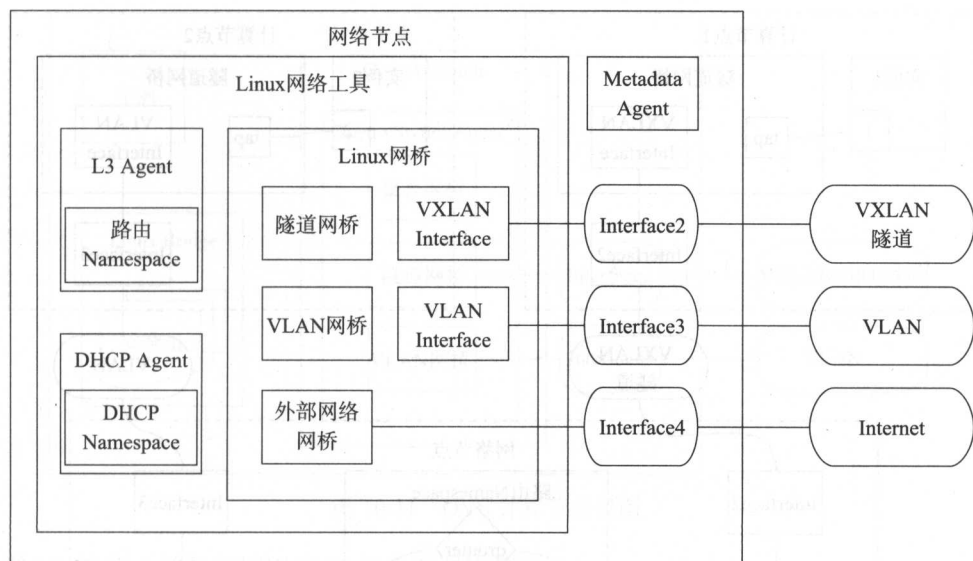


图 10-7 Linux Bridge 网络节点概图

然后实例与划分出的各种接口接驳，这些接口经过 Linux Bridge Agent 的部署之后，再与网络节点相连。Linux Bridge 计算节点概况如图 10-8 所示。

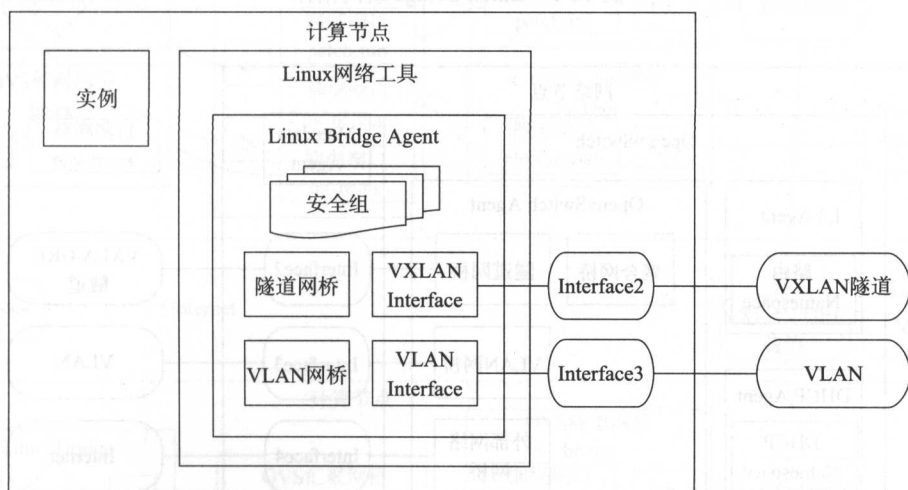
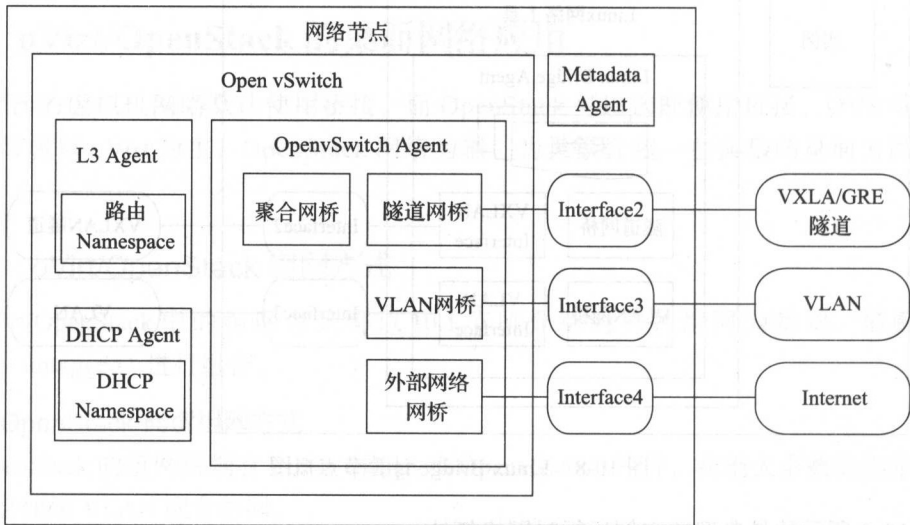
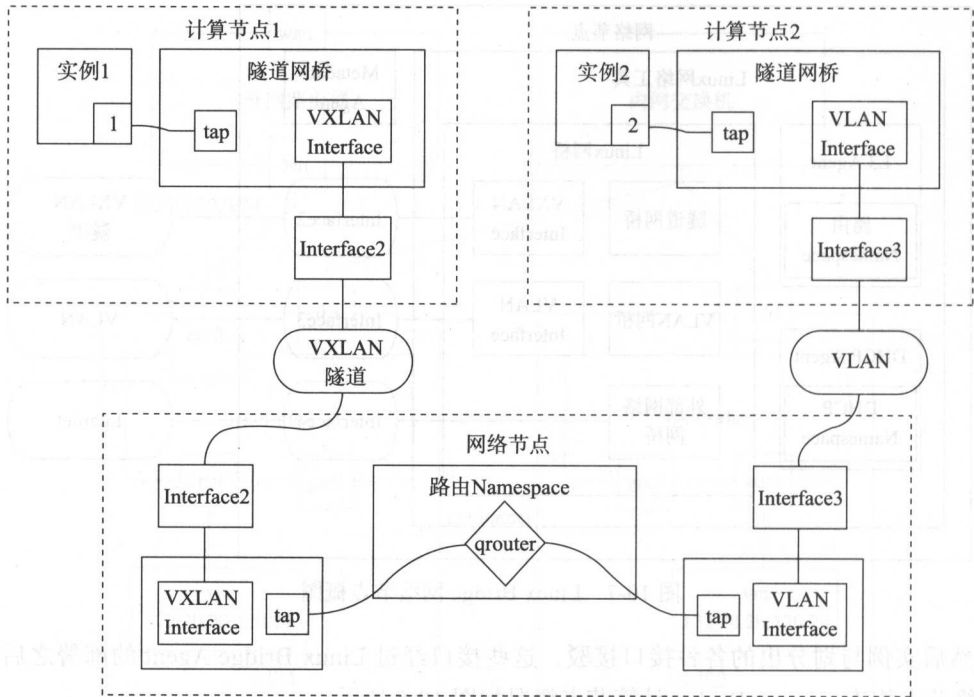


图 10-8 Linux Bridge 计算节点概图

图 10-9 所示的是典型的实例运行时网络拓扑：

## (2) OVS 组网

使用 Linux Bridge 作为 Neutron 后端时，网络节点（见图 10-10）会使用 OVS 命令完成所有三层、DHCP、交换功能，同时这些信息也会保留在元数据中。



实例在通信时，数据包经过管道、VLAN 标签处理之后，再经过网络节点 Flow Table 控制，最终数据得以传输。OVS 计算节点概况如图 10-11 所示。

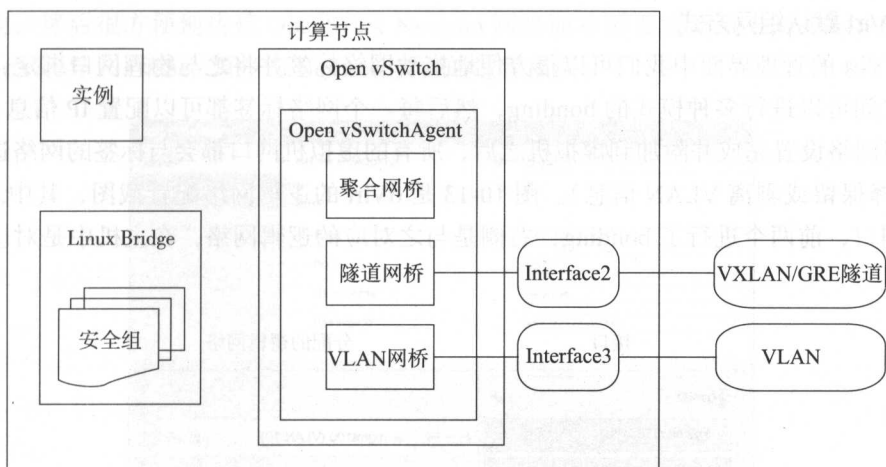


图 10-11 OVS 计算节点概图

图 10-12 所示的是使用固定 IP 的实例网络拓扑。

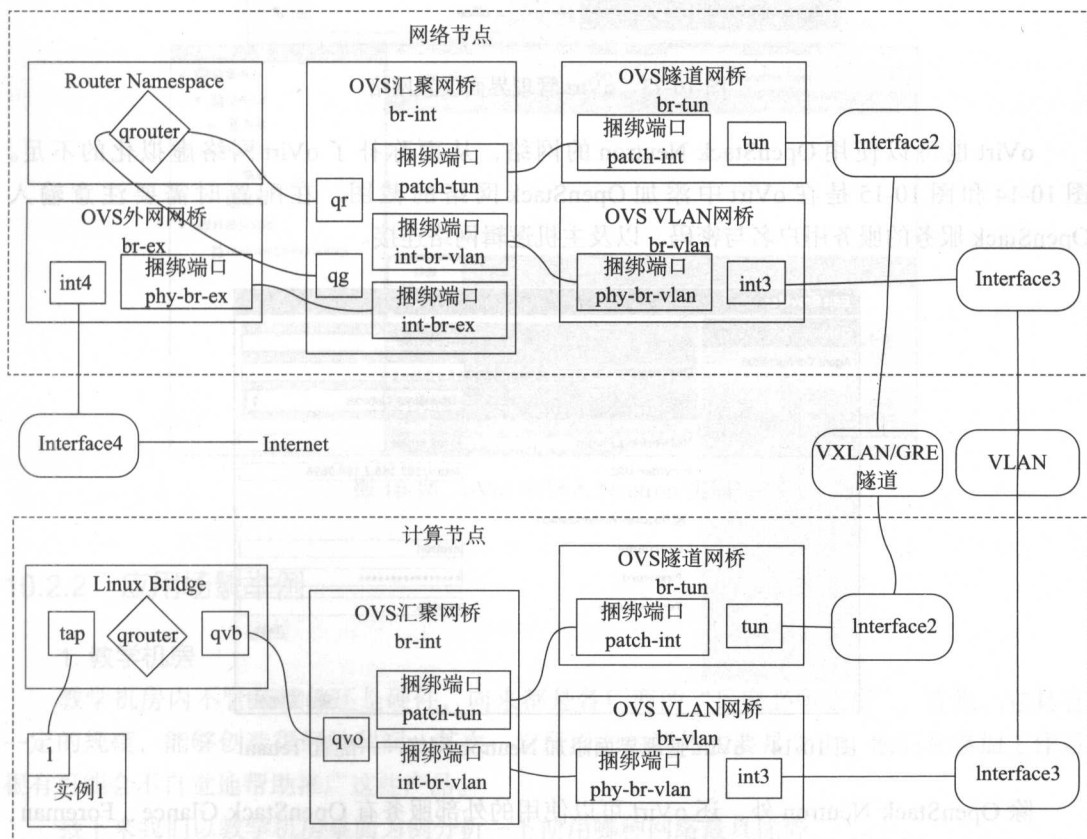


图 10-12 固定 IP 实例 OVS 网络拓扑



## 2. oVirt 默认组网方式

从 oVirt 的管理界面中我们可以很方便地拖动网络标签并将之与物理网口绑定，并且物理网口之间可以进行多种模式的 bonding，然后每一个网络标签都可以配置 IP 信息、VLAN 信息。当网络设置完成并附加到虚拟机之后，所有的虚拟机网口都会与标签的网络进行桥接（可以选择保留或剥离 VLAN 信息）。图 10-13 是 oVirt 的逻辑网络配置截图，其中左侧有 4 个物理网口，前两个进行了 bonding，右侧是与之对应的逻辑网络，在主机中是对应网桥的名称。

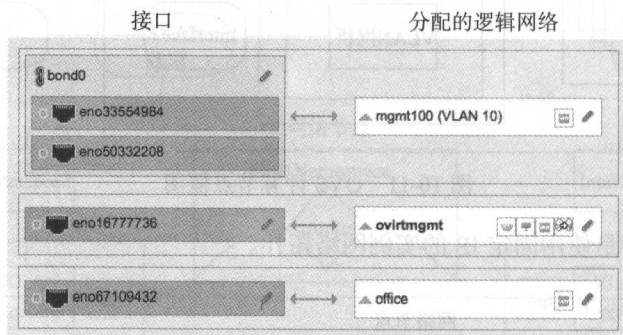


图 10-13 oVirt 管理界面设置网络

oVirt 也可以使用 OpenStack Neutron 的网络，从而弥补了 oVirt 网络虚拟化的不足。图 10-14 和图 10-15 是在 oVirt 中添加 OpenStack 网络的截图，在配置时需要注意输入 OpenStack 服务的服务用户名与密码，以及主机逻辑网络连接。

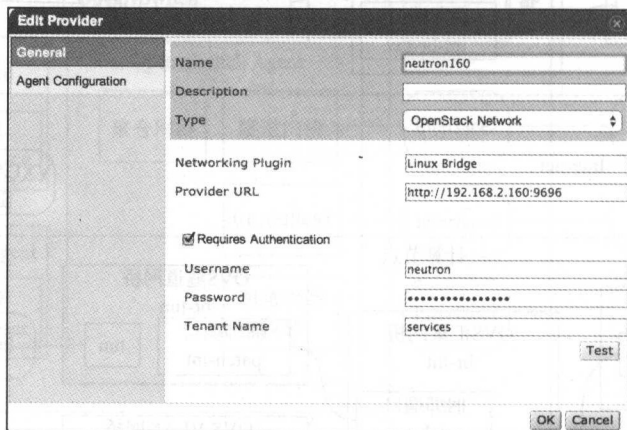


图 10-14 oVirt 管理界面添加 Neutron 网络——配置 Tenant

除 OpenStack Neutron 外，还 oVirt 可以使用的外部服务有 OpenStack Glance、Foreman、Docker 等。从其默认的 ovirt-image-repository 外部的 Glance 源中我们可以将 Neutron 实例导

入到本地，然后很方便地搭建 OpenStack Neutron 网络而不需要完整的 OpenStack 环境。图 10-16 是 oVirt 导入 Glance 源中的 Neutron 实例的过程截图。

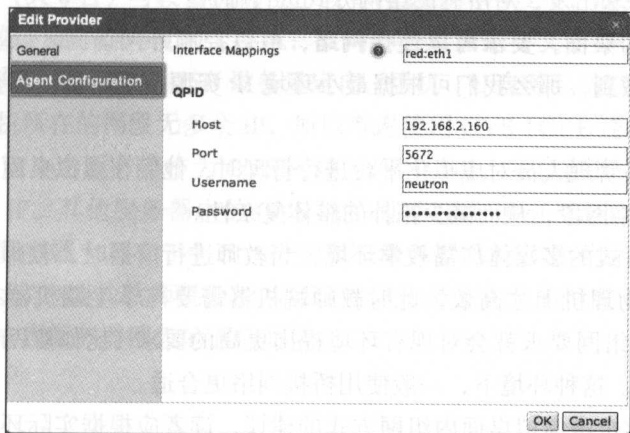


图 10-15 oVirt 管理界面添加 Neutron 网络——配置 MQ

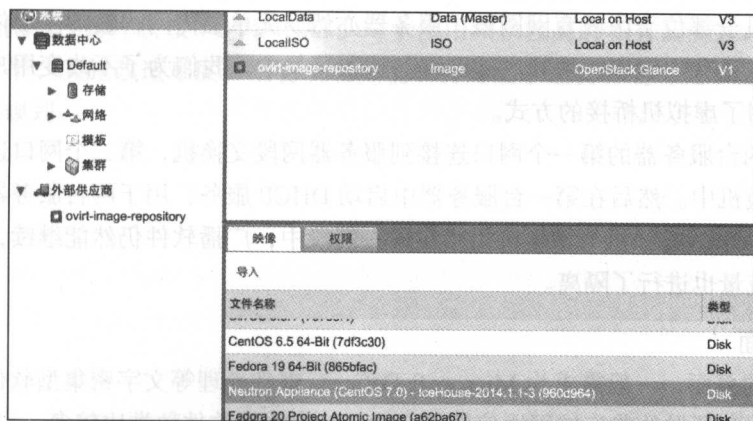


图 10-16 oVirt 中导入 Neutron 实例

## 10.2.2 应用场景举例

### 1. 教学机房

教学机房内不管是软件还是硬件，向来都是各厂商的“兵家必争之所”。首先，它具有一定的规模，能够创造很好的利润；其次，它所面向的群体主要是学生，他们在参加工作后很有可能不会自觉地帮助推广这些产品。

接下来我们以教学机房桌面为例分析一下使用哪种网络最具优势。

一般教学机房桌面，有安装软件繁多、使用时间固定、并发量大等特点，比较考验虚拟

化产品的综合素质。桌面安装软件除日常办公软件之外，还包括各种文字、图形密集类教学软件，同时还可能会安装影音广播教学类软件。如无特殊要求，很少会安装杀毒软件。

接下来我们从需求出发，对所有的组网方式进行筛选：

- ❑ 学生、教师的桌面，要求可以连接网络，可以控制外网访问。以上介绍的几种组网方式都可以做到，那么我们可根据最小环境 IP 资源占用原则，尽量减少桥接网络的使用。
- ❑ 教学机房网络管理人员对虚拟化平台进行管理时，他们在虚拟桌面中更倾向于分配简单的 NAT 网络网段，从而减少拓扑的整体复杂性。
- ❑ 对于纯软件方式的多媒体广播教学环境，当教师进行广播时，教师端编码往往在具有硬件显卡的物理机上才高效，此时教师端机器需要与学生端机器在同一网段，那么 NAT 或 OVS 组网要求就会对现有环境提出更高的要求（比如双网卡、服务器交换机额外 VLAN）。这种环境下，一般使用桥接网络更合适。

以上是我对教学机房虚拟桌面内组网方式的建议，读者应根据实际环境做出判断，在工作量与性能、功能间做出取舍。接下来我们以上海某中学的教学桌面云为例，介绍他们场景中的虚拟化网络部署。

其中学生机全部位于区教育网网段，服务器亦然，无单独内网；教师主要依赖桌面广播软件进行教学，且有比较多的视频广播课程。这种场景中，我们为了不改变用户原有的教学习惯，直接使用了虚拟机桥接的方式。

具体是将两台服务器的第一个网口连接到服务器网段交换机，第二个网口连接到教室所连接的上层交换机中。然后在第一台服务器中启动 DHCP 服务，用于两台服务器中虚拟机的 IP 分配。这样一来，教师机与学生机仍然在同一网段中，广播软件仍然能继续，且显示流量与虚拟机内部流量也进行了隔离。

## 2. 办公桌面

面向办公的桌面，一般需求为 Microsoft Office、邮件处理等文字密集型软件；通信类软件一般为国内厂商开发的非广域网通信软件及 QQ；防病毒软件种类比较多，目前卡巴斯基、诺顿等占多数，360 使用得较少；影音类软件的使用仅局限于网页 Flash，或者国内厂商定制的流媒体客户端。对于财务桌面，需求除普通办公桌面外还有一些财务类软件，而这些软件对桌面的负荷相较于普通办公桌面会高出一定量的资源消耗，同时也会有 U-Key、指纹仪等终端设备，所以对于这类桌面，我们一般进行特殊设置，比如将其固定到某台服务器上运行，并赋予一定优先级，保证资源优先分配。在普通桌面与财务桌面之外，还有浮动桌面可供出差人员或来访人员临时使用，此种桌面与一般的办公桌面无异，但可能要求有严格的用户检查控制及无状态模式要求，以防止恶意使用而导致损失。

以上就是办公桌面的典型场景，我们的组网方式也基本明确了。一般桌面使用 OVS 或 NAT，优先考虑 OVS；浮动桌面则使用 NAT；对于财务这类有特殊需求的桌面我们也可考虑使用桥接网络。另外当办公室中有类似打印机、扫描仪、指纹仪等共享设备时，它们的组网

优先选择桥接方式，这样便保证了所有人员都能对其进行访问。接下来以上海某高校的教师办公桌面云为例，介绍该场景中的网络部署。

其中，服务器（共 6 台）与教室机都位于教育网的不同网段中，服务器的 4 个网口进行 bonding 后连接到校园服务器网段的交换机；教师办公桌面需要文字办公软件、图形编辑软件、公式编辑软件等，并且支持 USB 设备重定向（手机、U 盘等）。这种场景中，由于作为客户端的原有教师机所在的网段无多余 IP，所以考虑使用 OVS（GRE 通道）方式进行组网。

我们选择了固定一台服务器作为 OVS 网段的外网出口，并且在其上启动了 DHCP 服务用于给虚拟机提供 IP，其他服务器中的 OVS 网桥通过 GRE 通道与出口服务器的 OVS 网桥相连。此时，出口服务器上具有 3 个 IP，分别属于服务器网段、显示网段、出口网段（用户网段）。最后，作为客户端的教师机便能够保证在显示带宽充足的情况下连接到云桌面，且其上网行为不会影响到服务器网段的通信质量。

### 10.3 小结

在平时的实施中，我们可以将组网按照 IP 资源稀缺性依次排列出 NAT>OVS> 桥接；同时按照访问权限范围排序桥接>OVS=NAT。对于使用频率较高的私有云而言，我们需要根据客户的运维水平、实施改造难度，结合现场环境和特殊需求进行综合考虑，这样才能得到比较满意的组网规划。

## 浅谈服务器交付的那些事儿

### 作者简介

赵旻, RHCA/RHCSS/MCITP, 熟悉 x86 平台基础架构系统的建设、管理及运维工作, 现就职于京东金融——网银在线, 担任支付产品技术部高级系统工程师一职; 9 年以上互联网金融、电信、政府等多领域背景的从业资历, 千万级大型项目经验, 优秀的文档撰写能力及沟通技巧; 曾参与中国国家电子政务多项重点工程的安全信任体系建设工作, 曾为中国移动、国航等大型企业提供运维技术支持; 喜欢从事新技术研究、优化方案等工作。工作前倾向于制定有计划且详细的执行方案, 擅长处理远期规划设计类事件。善于在工作实践中分析问题、总结经验, 属于改进优化能力类型的工作者。致力于对 IT 技术的精研和业务分析, 帮助企业定制更加实际高效的解决方案。

人们一提到 IT 运维, 必然离不开 IT 建设。随着市场化的不断壮大, 企业对设备的需求量也呈现出井喷式的增长。这对于运维来说, 海量的系统交付工作成为了一种严苛的考验。作为运维大军中的一个无名小卒, 我也经历过那样一段艰苦的岁月历程。

刚加入京东金融的时候, 当时的系统组只有两个人, 几百台服务器。我和另外一个负责网络的同事从无到有完成了两个 IDC 机房的建设工作。现如今, 京东金融已经实现了同城多活多机房的建设, 拥有了近万台服务器的规模。回想于此, 多少有些心得。因此我在这里撰写小文, 权当抛砖引玉, 以供大家参考。如有不到之处, 还望高人多多斧正。

### 11.1 设备签收的学问

签收地址的详细填写非常重要。我们采购的服务器非常多, 收货就是个大工程, 最少也

得几百台。有时候物流公司忙不过来，就会外包给第三方派送。而这些第三方有时会显得太过“精明”，他们很了解 IDC 机房会有专人负责拆箱上架的工作。如果收货地址写得不详细，他们把设备拉到楼下就算完成任务了。有些人甚至连卸货都不负责，还一个劲儿地催促你赶快签收。遇上这种情况将会很麻烦，因为 IDC 不是为你一家公司服务的，配备人员太少的话就会延误收货速度。另外，服务器不拆箱也没法正常验收（因物流原因造成损坏的设备应当拒收）。如果你把签收地址写到了机房里面，你就可以要求对方负责卸货并拆箱验收（这本来就是他们的分内工作），IDC 负责上架。这样我们就把人员全部都调动了起来，有效地提升了工作效率。签收地址在机房里面，只要设备没上架，你就有理由拒签。

对于设备的验收，一般我的做法是这样的：

（1）将设备卸到空场上（这需要 IDC 有一定的场地条件），要求物流按照箱体朝上、标签冲外的规则码放。如果外包装箱上有服务器型号，还可以分类码放。

（2）用记号笔在箱体上标注记号，清点设备数量。

（3）拆箱验货，验收完毕的设备准许上架。

在这个过程中有一个困难：我们有一些型号相同但配置不同的服务器。虽然标签上有配置清单，但要求物流按照配置清单进行分类就显得不太合理了，而且这些设备数量占比不大，也没法单独发送。因此我就养成了一个习惯：每次收货时都带着记号笔，在卸货的过程中，我按照配置自己做分类标记。卸货完成时，分类标记也就做完了。最后的数量清点工作就非常轻松了。

我曾经建议过厂商设置一个彩标方案，自定义一套彩色背景的标签。厂商和大客户签单时，在客户制订的类型与自定义色彩列表之间建立一个关系映射实例。例如 A 型设备对应红色标签，B 型设备对应黄色标签。在进行设备验收时，客户可根据标签颜色进行分类，从而有效提升签收效率，增强用户的体验效果，但是这个建议始终未被采纳。也许作为局外人，他们不曾有过这样的体会。一个人面对几百台型号各异的设备，核对配置是一件非常辛苦甚至是痛苦的事情。因为亲身经历过这种事，所以我也希望各大厂商今后能多一些同理心，多注重一些细节上的改善，做好客户服务才是与竞争对手拉开距离的制胜法宝。

## 11.2 服务器设置

服务器的有些配置可能无法预定义，所以你一定会遇到二次配置的场景。这里大体上可以分为 BIOS 配置、电源策略配置和带外管理配置三种类型。

对于带外管理地址的配置，通常有两种做法：

（1）事先规划各机架服务器的带外地址，设备上架后按照规划进行配置。

（2）提前向厂商索取带外管理卡的 MAC 地址表，采用 DHCP 地址绑定进行分配。

第一种方式的资产信息在部署前完成，设备排列规范整齐，便于机房人员的日常运维和



管理。适用于尚不具备完善有效的 CMDB 系统的场景。第二种方式的部署效率更高，还需要配合 IPMI 与 LLDP 完成数据采集的工作，将资产信息填写入库。但这也带来了一个比较大的挑战。设备是随机码放的，使用 DHCP 分配后的带外地址在机架上也是散列分布的。因此设备维护对 CMDB 系统有很强的依赖性，这就要求 CMDB 系统所采集的信息必须准确无误。其实现难度还是比较高的，里面存在着很多不确定因素的干扰（例如线路接错），会对数据采集产生较大的影响。因此对于程序的逻辑判断方面，有着非常复杂而严格的要求。因为当时整个运维部的成员不超过 20 个人，没有精力去做 CMDB 系统，所以早期我们采用的是第一种方式来实现。

但是，服务器设置工作也是一项繁重的劳动。纯手工操作的效率只有 100 台/人天，因此使用命令行接口来完成大批量修改的工作势在必行。本章将以 DELL 的 Rack 系列服务器为例向大家介绍一下如何进行带外管理的初始化。

准备一台装有 Linux 系统的笔记本，一根 30 米长的网线，设备正常加电，带外管理网络调通。参考命令如下，请注意执行的顺序。

### 1. 设置带外地址可用

具体的设置方法如下：

```
ipmitool -I lanplus -H 192.168.0.120 -U <USER> -P <PASSWORD> lan set <CHANNEL>
netmask <NETMASK>
ipmitool -I lanplus -H 192.168.0.120 -U <USER> -P <PASSWORD> lan set <CHANNEL>
defgw ipaddr <GATEWAY>
ipmitool -I lanplus -H 192.168.0.120 -U <USER> -P <PASSWORD> lan set <CHANNEL>
ipaddr <IPADDR>
```

注：192.168.0.120 是出厂的默认地址，只能接一台改一台，因此长网线可以让你在机房的正中间固定位置操作，而不必抱着笔记本跑来跑去。当然前提是有人能在另一端帮你插网线。^\_^

### 2. 修改带外管理的用户名和密码

具体的修改方法如下：

```
ipmitool -I lanplus -H <IPADDR> -U <USER> -P <PASSWORD> user set name <USER_ID>
<USER_NAME>
ipmitool -I lanplus -H <IPADDR> -U <USER> -P <PASSWORD> user set password
<USER_ID> <NEW_PASSWORD>
```

### 3. 开启 VNC 功能

具体的开启方法如下：

```
racadm set idrac.vncserver.enable enabled
racadm set idrac.vncserver.password <PASSWORD>
racadm set idrac.vncserver.port <PORT>
racadm set idrac.vncserver.timeout <SECOND>
```

注：DELL 的 VNC 功能非常棒，登录速度超快。另外 1.57 以后版本的 firmware 对花屏问题做了很大修缮，稳定性大大增加。建议把 timeout 的值调大一些。现在我已经把慢吞吞的虚拟控制台丢到一边去了。



#### 4. 其他注意事项

(1) 关于执行 `ipmitool` 命令时提示 “Unable to establish LAN session” 的问题。一般有两种情况。第一种，带外管理卡的问题。如果反复插拔网线不起效，可以关闭电源并拔掉电源线，然后按住电源按钮至少 30 秒进行放电操作。第二种，BIOS 设置问题，如果选项 “IPMI Over LAN” 没有启用也会触发这个错误。执行 `racadm set iDRAC.IPMLan.Enable Enabled` 即可。

(2) Boot Setting 的启动模式。Boot Setting 的默认值是 BIOS，但是有些做了 RAID 后磁盘空间大于 2TB 的机型，这个值有可能会被设置成 UEFI，从而导致 PXE 引导失败。大家可以看到图 11-1 所示的提示还是比较迷惑人的。

图 11-1 PXE 引导失败

解决方案是执行 `racadm set BIOS.BiosBootSettings.BootMode Bios` 进行调整。注意，命令执行完成后并不会立即生效，此时的键值只是一个待定值 (Pending Value)。需要创建一个 Job，然后重启服务器电源才能完成修改。创建 Job 的命令是 `racadm jobqueue create BIOS.Setup.1-1`。另外还可以使用命令 `racadm jobqueue view` 来查看 Job 的内容。

关于设置 Boot Setting 启动模式的完整过程。

##### ① 设置 Boot Setting 的启动模式为 BIOS：

```
racadm set BIOS.BiosBootSettings.BootMode Bios
```

##### ② 检查设置结果：

```
racadm get BIOS.BiosBootSettings.BootMode
/admin1-> racadm get BIOS.BiosBootSettings.BootMode
[Key=BIOS.Setup.1-1#BiosBootSettings]
BootMode=Uefi (Pending Value=Bios)
```

注意：此时只是 Pending Value，需要创建 Job 并重启主机电源。

##### ③ 创建 Job 检查无误后重启主机：

```
racadm jobqueue create BIOS.Setup.1-1
racadm jobqueue view
ipmitool -H <IPADDR> -U <USER> -P <IPADDR> power reset
```

##### ④ 重启完成后 Job 为完成状态，再次检查设置结果：

```
racadm jobqueue view
racadm get BIOS.BiosBootSettings.BootMode
```

(3) 关于电源策略的调整。电源策略有均衡和冗余两种模式。机柜电源 AB 两路本身是独立的，只要总体耗电不超过阈值，怎么使用那是用户的自由。不过调整成均衡模式对 IDC

会比较有利。就像你有两双鞋，两双来回倒换着穿，总比只啃着一双穿强。但要是你遇到那种强制要求用户改成均衡模式的 IDC，不妨多留个心眼儿，要小心他们的进线线径是否不够标准，或者是存在线路老化的问题。调整电源策略的命令如下：

```
racadm set System.Power.RedundancyPolicy 'Not Redundant'
racadm set System.Power.RedundancyPolicy 'Input Power Redundant'
```

## 11.3 Cobbler 的流程与规划

下面我们着重探讨一下 Cobbler 的安装流程和规划问题。Cobbler 各组件的关联结构如图 11-2 所示。

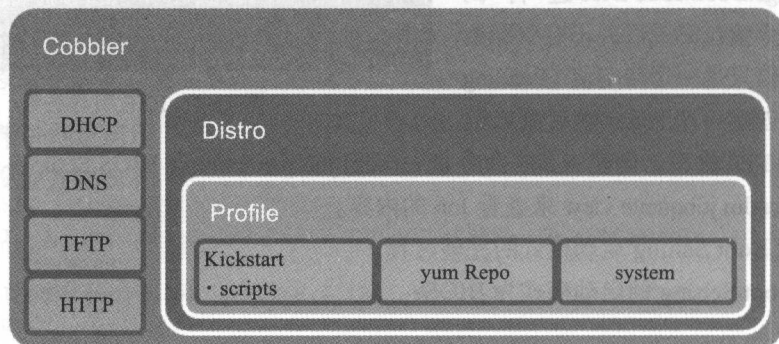


图 11-2 Cobbler 的组件架构图

鉴于 DNS 的特殊性，我们将其从 Cobbler 的管理中剥离出来，其他组件都交给 Cobbler 管理。前期已经对地址进行了分配，这里只需要在 Cobbler 中生成 system，最后通过 PXE 引导启动完成系统部署。生成 system 这一步比较麻烦，因此我写了一个小程序来完成这项工作。因为代码中有部分代码和逻辑涉及公司的敏感信息，不方便直接展示给各位读者。因此这里只把这个程序是如何实现的具体思路和几个关键命令列举出来。

实现思路如下：

- (1) 对于支持 WS-Management 标准的服务器，使用 `wsman` 命令直接采集网卡的 MAC 地址。
  - (2) 生成带外地址、业务地址和 MAC 地址之间的绑定关系，部署时 IP 就和设备对应起来了。
  - (3) 指定 TCP/IP 设置、主机名、Bonding 模式等，并据此生成创建 Cobbler system 的命令。
  - (4) Cobbler system 创建完成之后，使用 `ipmitool` 命令将下一次启动设置为由 PXE 引导。
  - (5) 使用 `ipmitool` 命令启动服务器开始安装部署。
- 关键命令如下：

```

# Collect mac address by ws-management.
wsman enumerate \
http://schemas.dmtf.org/wbem/wscim/1/cim-schema/2/root/dcim/DCIM_NICView \
-h <HOST> -V -v -c dummy.cert -P 443 -u <USER> -p <PASSWORD> \
-j utf-8 -y basic |grep PermanentMACAddress

# Add a system and bond master into cobbler.
cobbler system add --name=<NAME> --hostname=<HOSTNAME> \
--interface=<BOND_NIC> --interface-type=bond \
--bonding-opts="miimon=100 mode=4 xmit_hash_policy=layer2+3" \
--ip-address=<BOND_IP> --subnet=<NETMASK> --gateway=<GATEWAY> --static=1 \
--profile=<PROFILE>

# Append slave A to bond master.
cobbler system edit --name=<NAME> --mac==<MAC> \
--interface=<SLAVE1_NIC> --interface-type=bond_slave \
--interface-master=<BOND_NIC> \
--profile=<PROFILE>

# Append slave B to bond master.
cobbler system edit --name=<NAME> --mac==<MAC> \
--interface=<SLAVE2_NIC> --interface-type=bond_slave \
--interface-master=<BOND_NIC> \
--profile=<PROFILE>

# Set host next boot from pxe.
ipmitool -I lanplus -H <IPADDR> -U <USER> -P <PASSWORD> chassis bootdev pxe

# Reboot host.
ipmitool -I lanplus -H <IPADDR> -U <USER> -P <PASSWORD> power reset

```

## 拓展 1: 关于 Bonding 模式的选用

如果你研读一下《RHEL Deployment Guide》就可以得知, Bonding 模式一共有 7 种。从带宽利用率的角度来看, 肯定不会考虑模式 1 和模式 3。模式 0 存在数据包收发乱序的问题, 模式 2 仅仅基于二层作为定义域的 hash 算法对带宽的利用不够充分, 而模式 5 属于单向均衡。这样一来就只剩下 802.3ad 和 alb 两种模式了。alb 无须交换机配置, 采用 ARP 协商进行均衡, 每发送一次 ARP 请求广播或添加、删除一个 slave 节点的时候, 都需要重新协商分配流量, 而在协商之前所有流量都会涌向一个当前的 slave 端口。802.3ad 是一个业界标准, 通过创建一个聚合组, 确保组内所有链路的速率和工作模式一致, 这种方式对带宽的利用更为充分。关于 xmit\_hash\_policy 的选择我们使用了 layer2+3 的方式。layer3+4 的兼容性不好, 而 layer2+3 在默认 layer2 的基础上增加了对三层因素的考量, 分配会更加均衡。详细请参见 <https://www.kernel.org/doc/Documentation/networking/bonding.txt>。

## 拓展 2：关于 WS-Management 标准

WS-Management(全称为 Web Services-Management) 是 DMTF 组织基于 SOAP(Simple Object Access Protocol, 简单对象访问协议) 制定的一种开源标准, 该标准致力于在不同的 x86 设备厂商当中, 提供一种 IT 基础架构信息访问与修改的统一接口。凡是支持该标准的 x86 设备, 对于有效管理资产配置工作均提供了极大的帮助。例如, 我们有很多不同厂商的服务器, 如果它们都支持 WS-Management 标准, 那么我们就可以通过 wsmancli 工具, 去统一采集所有服务器的硬件配置信息, 也可以修改所有服务器的带外管理配置, 而不是使用各厂商的私有化工具去分别管理。像 AMD、DELL、Intel、Microsoft 这些知名厂商都是该项标准组的成员。当然最新的类似标准还有 redfish, 支持的成员更多, 但 redfish 目前还没有推广开来。

有兴趣的读者请参见 <http://www.dmtf.org/standards/wsman> 和 <http://www.dmtf.org/standards/redfish>。

在谈部署架构的规划之前, 我们先看一下 Cobbler 体系架构中各组件的作用。Distro 指的是操作系统的分发版本。如果你的生产环境需要 CentOS 5、CentOS 6 和 RHEL 7 三个版本, 那么你就需要创建三个 Distro。Distro 下面包含了 Profile, 你可以认为 Profile 就是你的一个部署模板, 不同的 Profile 之间的差异应该包括 RAID 配置、分区和软件包。如果你要用 CentOS 6 部署 MySQL 数据库和 Web Server, 很显然这两者的安装要求应该是不一样的, 你应当为此创建两个不同的 Profile。每一个 Profile 都对应了 kickstart、yum repo 和 system, 这里重点讲一下 system 的概念, system 里面包含了一台实际需要部署的服务器的相关系统信息, 这些信息包括主机名、TCP/IP 设置、网卡 Bonding 模式、VLAN 配置、使用哪个操作系统(由 Profile 对应的 Distro 来决定)、使用哪些 yum 源(由 Profile 对应的 yum repo 来决定)及部署成什么样子(由 Profile 对应的 Kickstart 来决定)等。按照面向对象的讲法, 如果说 Profile 是类的话, 那么一个 system 就是一个部署实例。如果你用名为 MySQL 的 Profile 安装了 200 台主机, 那么就要创建 200 个 system。

对于部署架构的规划, 我会根据现有业务场景的需要, 针对不同场景定制不同的 Distro 和 Profile, 每个 Profile 对应自己的 Kickstart。具体实现还是以 MySQL 和 Web Server 的部署来举例: 首先做一个最小化系统安装的模板——名为 Base 的 Profile 和名为 base.ks 的 Kickstart, 在 base.ks 的基础上添加 MySQL 需要安装的软件包并修改分区设置, 并为此创建 MySQL 和 mysql.ks, 照方抓药再完成 Web 和 web.ks 即可。我希望 Kickstart 文件只描述两个内容——分区设置与安装所需的软件包, 其他的调整均通过脚本实现。

对于大批量部署还有一种见解不同的做法, 那就是采用镜像部署方案。这种方案倾向于制作一个涵盖所有场景需求的镜像文件, 部署时把这个镜像文件推送并解压到系统上, 最后用脚本修改差异化配置。

两种方案各有优势。第一种方案如果发生了配置变更，修改内容越靠近分支，其修改成本就越低，其影响范围是可控的。而且不受多场景差异化的干扰，不必刻意考虑融合所带来的种种兼容性问题。第二种方案并发能力更强，交付速度更快。但是我个人不太欣赏这种做法。尤其是在对现有模板进行软件包删除、Patch 更新或新驱动编译的时候，繁琐的封包操作真的是非常麻烦。而且我也不认为前者的交付速度会影响到业务。实测过 200 以上的并发是没有任何问题的，基本上一轮部署也就是几分钟的事情。照此计算，如果不算机房的前置工作，一天至少也可以交付 5000 台左右。与其斤斤计较两种部署方式所相差的那一两分钟，还不如去做好业务申请及交付的流程优化来得更加实在。

关于 %post 脚本的设计，我仅仅使用 curl 调用的方式来实现。例如有一个场景 A 需要交付，我利用 curl 调用脚本 Profile\_A，而 Profile\_A 里面也只是利用 curl 调用二级脚本 Detail01、Detail02、Detail03 等。对于场景 B 的交付如法炮制。也就是说，每一个 Profile\_X.ks 都对应于一个 Profile\_X，Profile\_X 又对应于若干个 DetailXX，而 DetailXX 才是真正完成配置修改的部分。在这里我把所有的 Profile\_X 都放置到一个名为 /opt/scripts/post/ 的子目录中，把所有的 DetailXX 放置到一个名为 /opt/scripts/post.d/ 的子目录中。相信大家也看懂了，这种方式其实就是效仿系统 /etc/rcX.d/ 和 /etc/init.d/ 的关系建立的。当进行系统调试的时候，可以直接修改脚本，运行一下就能看到效果，而不是重新 kick 一遍主机。

## 11.4 服务器安装时遇到的各种坑

每次做批量部署，在安装第一台主机时总是会不顺利，遇到的问题也是千奇百怪。下面我就把这些问题的解决方法分享给大家。

### 11.4.1 DHCP 客户端获取 IP 地址失败

这类问题基本上都出现在跨网段的场景，如图 11-3 所示。

```
Scanning for devices. Please wait, this may take several minutes...

Intel(R) Boot Agent GE v1.4.03
Copyright (C) 1997-2012, Intel Corporation.

CLIENT MAC ADDR: EC F4 BB C3 3C E0 GUID: 44454C4C 5800 1037 8051 B3C04F573232
PXE-E51: No DHCP or proxyDHCP offers were received.

PXE-M0F: Exiting Intel Boot Agent.
No operating system is currently installed on this computer._
```

图 11-3 DHCP 客户端获取 IP 地址失败的场景

我总结了以下几种情况：

- 某品牌交换机需要禁用 STP，否则收不到 DHCP 请求（据说后来升级 firmware 就解决了）。
- 对 VRRP 支持不好，网关不接收广播报文，尝试禁用或使用 HSRP。



□ 网卡模块损坏。

□ 此外还有一种情况是插错网线所导致的，PXE 在启动时会提示 “media failure”。

### 11.4.2 TFTP 加载失败

TFTP 可用于加载 pxelinux.cfg、vmlinuz 和 initrd.img 的文件传输服务。这道关卡的故障处理难度也比 DHCP 的环节增加了不少。



**注意** 因为以下这些截图全部来自于我们的生产环境，所以我针对部分敏感信息做了遮蔽处理，请各位读者见谅。

没有开启 ARP 代理，访问 TFTP 服务时会提示 “ARP timeout”，如图 11-4 所示。

```
Copyright (C) 1997-2012, Intel Corporation
CLIENT MAC ADDR: EC F4 BB C3 0C B4 GUID: 44454C4C 3300 104A 8034 C8C04F3B3232
CLIENT IP: MASK: DHCP IP:
PXE-E11: ARP timeout
PXE-MOF: Exiting Intel Boot Agent.

Intel(R) Boot Agent GE v1.4.03
Copyright (C) 1997-2012, Intel Corporation
CLIENT MAC ADDR: EC F4 BB C3 0C B4 GUID: 44454C4C 3300 104A 8034 C8C04F3B3232
CLIENT IP: MASK: DHCP IP:
PXE-E11: ARP timeout
PXE-MOF: Exiting Intel Boot Agent.

No boot device available.
Current boot mode is set to BIOS.
Please ensure compatible bootable media is available.
Use the system setup program to change the boot mode as needed.

Strike F1 to retry boot, F2 for system setup, F11 for BIOS boot manager.
```

图 11-4 TFTP 加载失败案例一

### 11.4.3 TFTP Client 交互后无响应

这种倒霉事我遇到过两次，而且还是出自两个不同的原因。第一次是在安装物理机时，TFTP 运行了一下就退出了，如图 11-5 所示。

```
Strike F1 to retry boot, F2 for system setup, F11 for BIOS boot manager.

Intel(R) Boot Agent GE v1.5.56
Copyright (C) 1997-2014, Intel Corporation
CLIENT MAC ADDR: EC F4 BB C3 C0 DC GUID: 44454C4C 3600 104B 8059 C8C04F5A3232
CLIENT IP: MASK: DHCP IP:
GATEWAY IP:
TFTP.
```

图 11-5 TFTP 加载失败案例二

messages 日志提供不了太多的帮助，我通过使用 tcpdump 抓包来进行故障分析，结果在里面看到了这样一行——缺少 filename “undionly.kpxe”。这个 undionly.kpxe 看着非常眼熟，不正是配置文件 /etc/cobbler/dhcp.template 中的 if \$iface.enable\_gppe 部分么？原来是某同事在测试 VM 安装时启用了 gppe，后来又忘记关掉了。注意，修改了 /etc/cobbler/setting 中的 enable\_

gppe: 0 后还不算完, 因为 system 已经生成了, 所以需要更新配置 system 或重建 system。

第二次是一上来就直接报 “TFTP open timeout”, 如图 11-6 所示。

```
Intel(R) Boot Agent GE v1.5.56
Copyright (C) 1997-2014, Intel Corporation

CLIENT MAC ADDR: EC F4 BB C3 C0 DC  GUID: 44454C4C 3600 104B 8059 C8C04F5A3232
CLIENT IP:  MASK:  DHCP IP: 
GATEWAY IP: 
PXE-E32: TFTP open timeout
PXE-M0F: Exiting Intel Boot Agent.

No boot device available.
Current boot mode is set to BIOS.
Please ensure compatible bootable media is available.
Use the system setup program to change the boot mode as needed.

Strike F1 to retry boot, F2 for system setup, F11 for BIOS boot manager.
```

图 11-6 TFTP 加载失败案例三

当时使用的是 Cobbler 2.6.3, 在进行 get-loaders 的时候, 被系统告知部分 loader 文件无法下载。因为我手上有一份备份, 对此也没在意, 就只复制了失败的部分, 结果后面就出现了这个悲剧。

tcpdump 抓包也只看到 Extra Error, 再往后就语焉不详了。于是, 我冷静下来反思系统安装的流程。当想到 TFTP 本来就是要提供 loader 文件时, 自己一下子就释然了。使用 md5sum 命令很快就印证了 loader 文件传输损坏的想法。我把所有 loader 文件重新复制了一份, 故障自然解除。

#### 11.4.4 yum 安装失败

当安装过程中报告文件缺失或损坏的时候 (如图 11-7 所示), 你可别天真地认为是镜像有问题。检查一下日志输出, 有可能是网络断了。切换到 shell 后用 route 和 ip addr 去判断。还有一种情况是我同事碰到的。他把 update 的软件包也放到 yum 源里面了, 但要更新的包不属于原生的 yum 源, 结果导致 yum 在计算依赖关系时出现了逻辑错误, 产生了一个先有鸡还是先有蛋的悖论问题。解决的办法就是把更新部分放到 %post 里面去, 或者到网站去更新最新版本的 repo 文件。

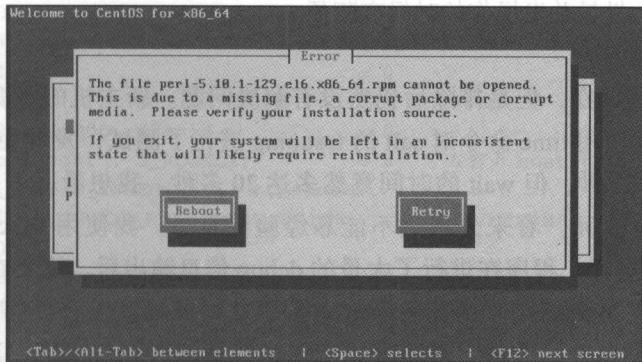


图 11-7 yum 安装失败



### 11.4.5 Linux 内核无法识别新硬件

如果 Linux 内核无法识别新硬件（例如 RAID 卡）则有可能导致安装失败。采用镜像部署方案的小伙伴们会非常痛苦。因为你需要重新编译内核，还要不断地进行封包测试。对于采用 Cobbler 的小伙伴们就有福了。首先找到硬件驱动的 rpm 包，将其制作成 ISO 放到部署服务器的 Web 服务上面，然后在 kickstart 文件里找到关于分区描述的地方，在其下添加这样一条语句：

```
driverdisk --source=http://x.x.x.x/newdriver.iso
```

如此一来，就可以在开机时自动加载驱动光盘了。

### 11.4.6 恶意 PXE 启动导致原有系统被误装

如果在维修过程中遇到主板损坏的服务器一定要非常小心，因为新更换主板的 BIOS 设置有可能是 PXE 启动，如果不注意贸然开机将导致 PXE 引导触发系统重装的悲剧。将 Cobbler 主配置文件里面的 pxe\_just\_once 选项设置为 1 并重启 cobblerd 服务即可确保以后生成的 system 只响应一次 PXE 引导请求。对于再此之前已经生成或安装完毕的 system 强烈建议使用如下命令来禁止 Cobbler 响应 PXE 引导请求。

```
cobbler system edit --name=<NAME> --netboot-enabled=False
```

## 11.5 交接后的故事

后来团队的规模慢慢壮大起来了，我就把这份“前线差事”交付给了其他同事。原以为“艰苦岁月”总算告以段落，谁想清静了没两个月，又冒出了一个新问题。

一位监控同事找到我，说昨天新交付的一批主机要部署 Check\_MK，但是用 Salt-Master 批量推送时特别慢。我当时觉得问题很简单，无非就是如下几种可能：

- ❑ 群组里面存在状态为 down 的 minion 节点。
- ❑ Salt-Master 大批量并发操作的时候有瓶颈。
- ❑ 资源占用问题。

第一种情况确实存在，但是排除后仍然没有解决问题。新主机的资源占用很低，而且只有 salt 命令慢。我使用 time 命令对 salt 的 test.ping 进行了测试，发现 usr 和 sys 所消耗的时间加起来也不过 0.2 秒，但 wait 的时间竟然多达 20 多秒。我想看一下版本号，哪知 salt --version 也同样慢得要死。看来人永远不能靠经验混日子。我使用 strace 命令去跟踪 salt --version 运行的整个过程，程序在进行了大量的 debug 信息输出后，终于在某个时刻停了下来。这个关键时段就是我们刚才所谓的 wait 时间，此时屏幕上没有任何输出。趁此机会，我赶紧截取了当前屏幕上的内容，然后等待着程序继续运行直至完毕。当 strace 执行完成后，我把这个命令又重复执行了一次。但与上一次不同的是，这次我把输出的结果保存到了

strace.log 文件中以便于分析。根据第一次程序暂停时所截取的屏幕内容，我在 strace.log 中找到了它的位置。这段代码是标准的 socket 网络通信连接的过程，重点关注 connect() 函数提交的 sin\_port 和 sin\_addr 这两个参数，很显然这是一个 DNS 查询：

```
socket(PF_INET, SOCK_DGRAM|SOCK_NONBLOCK, IPPROTO_IP) = 3
connect(3, {sa_family=AF_INET, sin_port=htons(53), sin_addr=inet_addr("x.x.x.x")}, 16) = 0 // 问题就出在这里
poll([{fd=3, events=POLLOUT}], 1, 0) = 1 ([{fd=3, revents=POLLOUT}])
sendto(3, "~\347\1\0\0\1\0\0\0\0\0\0\0\0\fexample\1h\tchin"... , 68, MSG_NOSIGNAL, NULL, 0) = 68
poll([{fd=3, events=POLLIN|POLLOUT}], 1, 3000) = 1 ([{fd=3, revents=POLLOUT}])
sendto(3, "\311\333\1\0\0\1\0\0\0\0\0\0\0\0\fexample\1h\tchin"... , 68, MSG_NOSIGNAL, NULL, 0) = 68
poll([{fd=3, events=POLLIN}], 1, 2999) = 0 (Timeout)
socket(PF_INET, SOCK_DGRAM|SOCK_NONBLOCK, IPPROTO_IP) = 4
connect(4, {sa_family=AF_INET, sin_port=htons(53), sin_addr=inet_addr("y.y.y.y")}, 16) = 0 // 问题就出在这里
poll([{fd=4, events=POLLOUT}], 1, 0) = 1 ([{fd=4, revents=POLLOUT}])
sendto(4, "~\347\1\0\0\1\0\0\0\0\0\0\0\0\fexample\1h\tchin"... , 68, MSG_NOSIGNAL, NULL, 0) = 68
poll([{fd=4, events=POLLIN|POLLOUT}], 1, 6000) = 1 ([{fd=4, revents=POLLOUT}])
sendto(4, "\311\333\1\0\0\1\0\0\0\0\0\0\0\0\fexample\1h\tchin"... , 68, MSG_NOSIGNAL, NULL, 0) = 68
poll([{fd=4, events=POLLIN}], 1, 5999) = 1 ([{fd=4, revents=POLLERR}])
close(3) = 0
close(4) = 0
brk(0x11f3000) = 0x11f3000
```

x.x.x.x 和 y.y.y.y 是我们的两台生产 DNS Server，而这台 Salt-Master 是访问不了的。我们再向上翻看，确实有读取 nsswitch.conf，hosts.conf 和 resolv.conf 等文件内容的操作，代码如下所示：

```

open("/etc/nsswitch.conf", O_RDONLY)      = 3      // 打开 nsswitch.conf 文件
fstat(3, {st_mode=S_IFREG|0644, st_size=1688, ...}) = 0
mmap(NULL, 4096, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) =
0x7fd2dac4a000
read(3, "#\n# /etc/nsswitch.conf\n#\n# An ex"... , 4096) = 1688
read(3, "", 4096) = 0
close(3) = 0
munmap(0x7fd2dac4a000, 4096) = 0
open("/etc/host.conf", O_RDONLY) = 3      // 打开 host.conf 文件
fstat(3, {st_mode=S_IFREG|0644, st_size=9, ...}) = 0
mmap(NULL, 4096, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) =
0x7fd2dac4a000
read(3, "multi on\n", 4096) = 9
read(3, "", 4096) = 0
close(3) = 0
munmap(0x7fd2dac4a000, 4096) = 0
futex(0x30d5d91384, FUTEX_WAKE_PRIVATE, 2147483647) = 0

```

```

open("/etc/resolv.conf", O_RDONLY) = 3 // 打开 resolv.conf 文件
fstat(3, {st_mode=S_IFREG|0644, st_size=194, ...}) = 0
mmap(NULL, 4096, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) =
0x7fd2dac4a000
read(3, "# Generated by Cobbler post_scri"... , 4096) = 194
read(3, "", 4096) = 0
close(3) = 0

```

我们再向上翻看，观察程序运行初始化的过程，代码如下所示。

```

open("/etc/salt/minion_id", O_RDONLY) = -1 ENOENT (No such file or directory)
uname({sys="Linux", node="stationX.example.com", ...}) = 0
socket(PF_NETLINK, SOCK_RAW, 0) = 3
bind(3, {sa_family=AF_NETLINK, pid=0, groups=00000000}, 12) = 0
getsockname(3, {sa_family=AF_NETLINK, pid=34099, groups=00000000}, [12]) = 0
sendto(3, "\24\0\0\0\26\0\1\3\1q\304U\0\0\0\0\0\0\0", 20, 0, {sa_family=AF_
NETLINK, pid=0, groups=00000000}, 12) = 20
recvmsg(3, {msg_name(12)={sa_family=AF_NETLINK, pid=0, groups=00000000}, msg_iov(1)=
[{"0\0\0\0\24\0\2\0\1q\304U\3\205\0\0\2\10\200\376\1\0\0\0\10\0\1\0\177\0\0\1"... ,
4096}], msg_controllen=0, msg_flags=0}, 0) = 108
recvmsg(3, {msg_name(12)={sa_family=AF_NETLINK, pid=0, groups=00000000}, msg_iov(1)
=[{"@0\0\0\24\0\2\0\1q\304U\3\205\0\0\n\200\200\376\1\0\0\0\24\0\1\0\0\0\0\0"... ,
4096}], msg_controllen=0, msg_flags=0}, 0) = 128
recvmsg(3, {msg_name(12)={sa_family=AF_NETLINK, pid=0, groups=00000000}, msg_iov
(1)=[{"\24\0\0\0\3\0\2\0\1q\304U\3\205\0\0\0\0\0\0\1\0\0\0\24\0\1\0\0\0\0\0"... ,
4096}], msg_controllen=0, msg_flags=0}, 0) = 20
close(3) = 0

```

Salt-Master 运行任何 salt 命令之前，都会先取得主机的信息并尝试进行名称解析，因为 /etc/resolv.conf 文件中配置了不可访问的 DNS Server，而且在 resolv.conf 文件中关于超时的设置没有明确定义，该设置项的默认值为 10 秒，两个不能访问的 DNS 各自都等待了 10 秒钟，最终引发了这个 wait 时间长达 20 多秒的问题。删除了 /etc/resolv.conf 中的错误条目后，故障立即解除。

这个解决问题的思路很巧妙。首先利用 time 观察到了时间瓶颈，既然是 wait，就说明程序在这段时间除了等待什么都没干。而 strace 的作用就是把程序运行时的细节展示到前台上来，那么在 wait 时段的 Debug 信息肯定也是停止输出的，相当于程序自动设置了一个断点。我们第一次操作的目的是取得断点内容，第二次操作的目的是为了拿到完整的 Debug 信息。最后根据断点内容在整个 Debug 信息中找到其所在的位置，该断点位置附近的上下文中一定隐藏着真正的故障原因。

当然，最后我们还要提醒一下负责部署的同事，对相关问题进行同步修改。

## 11.6 小结

想要做好基础架构的工作，除了要掌握应有的技术之外，我认为更重要的是以下几点：

(1) 要有全局的观念, 考虑问题不要总是从自我利益出发。在技术的选用上, 只关心如何让自己的工作更轻松, 却不管别人的死活。不要坐了架构的位置, 思想还停留在工程师的水平。

(2) 要注重细节, 不要做浮云架构, 光会画大饼是不成的。做任何设计一定要考虑能否落地? 自己是否亲自实践过? 我觉得每个架构师都应该背着自己设计的降落伞至少跳一次飞机, 而不是用别人的生命和鲜血去验证结果。

(3) 杜绝无底线的服务意识, 实际业务需求和整体基础架构规范应当和谐共存, 不能相互绑架。

(4) 不要觉得自己了不起, 应当虚心并且不断地学习, 多向别人讨教, 因为总有你不会的东西, 即便是你认为再简单的事情。

(5) 有了成绩要多分享不要舍不得(超级解霸作者梁肇新的话)。当然, 请一定要分享干货, 也就是有细节、能落地、经得起验证的真成绩。

(6) 要注意技术和知识的灵活运用, 11.5 节 `strace` 命令的使用就是一个好例子。

(7) 要有一个可靠的伙伴, 我的同事张望在网络方面给予了我很多支持, 他是这方面的技术专家, 能够快速定位并解决部署过程中遇到的各种奇葩故障, 而且还是一个很帅气的小伙儿。

(8) 要有一位能认可你的能力、支持你的想法、并且包容你犯错误的好领导, 这条最重要。

关于服务器交付的事儿, 我们就聊到这里。有些读者可能会觉得 SA 不就是一个装机器的么? 但基础架构无小事, 即便是小小的装机, 也同样严苛地考验着一个 SA 的技术水平。要知道, 安装部署和服务配置只是 SA 的基本工作, 能够真正地深入系统核心, 透析系统机制, 胜任内核参数调整、故障快速定位、解析 `CoreDump` 以及编写内核代码的工作, 才是 SA 应有的境界和高度。

## 企业级 Nginx Web 服务优化实战

### 作者简介

冉宏元（老男孩），北京老男孩 IT 教育创始人，拥有 10 多年一线大规模网站集群实战运维架构经验及教学培训经验，经历并主导了服务器从几台到近千台大规模集群运维架构的发展过程；运维架构实战知识体系全面，擅长大规模集群架构部署调优、虚拟化、云计算、大数据及 MySQL 数据库等技术，是 IT 界最资深的 Linux 集群架构实战专家之一。

国内 NLP 心理学运维思想体系创始人，将心理学运维思想大量应用于教学培训实践，取得了显著效果，所教学生平均就业工资及后期发展速度连续多年在国内同行业排名第一！

## 12.1 Nginx 基本安全优化

### 12.1.1 调整参数隐藏 Nginx 软件版本号信息

一般来说，软件的漏洞都和版本有关，这一点很像汽车的缺陷，同一批次的产品要有问题就都有问题，别的批次可能就都是好的。因此，我们应尽量隐藏或消除 Web 服务对访问用户显示各类敏感信息（例如 Web 软件名称及版本号等信息），这样恶意的用户就很难猜到他攻击的服务器所用的是否有特定漏洞的软件，或者是否有对应漏洞的某一特定版本，从而加强了 Web 服务的安全性。这在武侠小说里，就相当于隐身术，你隐身了，对手就很难打着你了。

想要隐身，首先要了解所使用软件的版本号，对于 Linux 客户端，可通过命令行查看 Nginx 版本号，最简单的方法就是在 Linux 客户端系统命令行执行如下 curl 命令：

```
[root@goldboy ~]# curl -I 10.0.0.7
```

```

HTTP/1.1 200 OK
Server: nginx/1.6.3    #<== 这里很清晰地暴露了 Web 版本号 (1.6.3) 及软件名称 (Nginx)
Date: Thu, 09 Oct 2014 01:58:51 GMT
Content-Type: text/html
Content-Length: 18
Last-Modified: Thu, 25 Sep 2014 20:01:01 GMT
Connection: keep-alive
ETag: "5424747d-12"
Accept-Ranges: bytes

```

在 Windows 客户端上, 通过浏览器访问 Web 服务时, 若找不到页面, 那么默认的报错信息如图 12-1 所示:

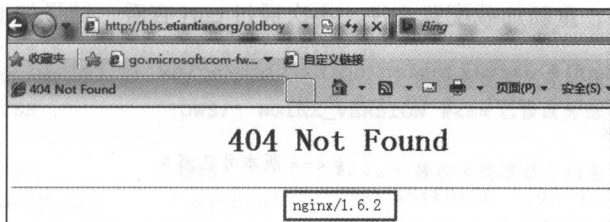


图 12-1 找不到对应地址的错误页面显示

以上虽然是不同的客户端, 但是都获得了 Nginx 软件名称, 而且查到了 Nginx 的版本号, 这就使得 Nginx Web 服务的安全存在一定的风险, 因此, 应隐藏掉这些敏感信息或用另一个其他的名字将其替代。例如, 下面是百度搜索引擎网站 Web 软件的更名做法:

```

[root@oldboy ~]# curl -I baidu.com
HTTP/1.1 200 OK
Date: Tue, 21 Oct 2014 03:31:01 GMT
Server: Apache    # <== 将 Web 服务软件更名为 Apache, 并且版本号也去掉了
[root@oldboy ~]# curl -I -s www.baidu.com | grep Server
Server: BWS/1.1    # <== 将 Web 服务软件更名为 BWS, 并且版本号被改为 1.1 (闭源的软件名称和版本就无所谓了)

```

门户网站尚且如此, 我们也学着隐藏或改掉应用服务的软件名和版本号吧!

事实上, 还可以通过配置文件加参数来隐藏 Nginx 版本号。

编辑 nginx.conf 配置文件增加参数, 实现隐藏 Nginx 版本号的方式如下。

在 Nginx 配置文件 nginx.conf 中的 http 标签段内加入 “server\_tokens off;” 参数, 具体如下:

```

http
{
    .....
    server_tokens off;
    .....
}

```



此参数放置在 `http` 标签内，作用是控制 `http response header` 内的 Web 服务版本信息的显示，以及错误信息中 Web 服务版本信息的显示。

`server_tokens` 参数的官方说明如下：

```
syntax:    server_tokens on | off; # <== 此行为参数语法, on 为开启状态, off 为关闭状态
default:   server_tokens on;      # <== 此行的意思是不配置该参数, 软件默认情况的结果
context:   http, server, location # <== 此行为 server_tokens 参数可以放置的位置
参数作用: 激活或禁止 Nginx 的版本信息显示在报错信息和 Server 的响应首部位置中
Enables or disables emitting of nginx version in error messages and in the "Server"
response header field.          # <== 此行是参数的作用原文, 一定要细看
```

官方资料地址: [http://nginx.org/en/docs/http/nginx\\_core\\_module.html](http://nginx.org/en/docs/http/nginx_core_module.html)。

配置完毕后保存，重新加载配置文件，再次通过 `curl` 查看，结果如下：

```
[root@oldboy conf]# /application/nginx/sbin/nginx -s reload
[root@oldboy conf]# curl -I 10.0.0.7
HTTP/1.1 200 OK
Server: nginx # <== 版本号已消失
Date: Thu, 09 Oct 2014 02:03:32 GMT
Content-Type: text/html
Content-Length: 18
Last-Modified: Thu, 25 Sep 2014 20:01:01 GMT
Connection: keep-alive
ETag: "5424747d-12"
Accept-Ranges: bytes
```

此时，浏览器的报错提示中没有了版本号，如图 12-2 所示，修改成功。

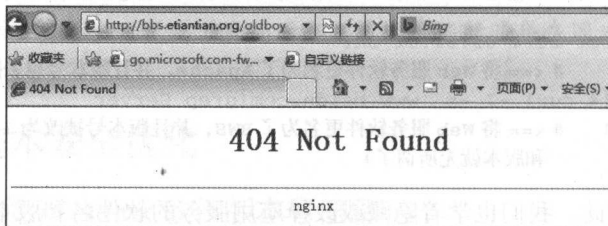


图 12-2 无版本号的错误页面显示

### 12.1.2 更改源码隐藏 Nginx 软件名及版本号

隐藏了 Nginx 版本号后，更进一步，可以通过一些手段把 Web 服务软件的名称也隐藏起来，或者更改为其他 Web 服务软件名以迷惑黑客。但对于软件名字的隐藏修改，一般情况下不会有配置参数和入口，Nginx 也不例外，这可能是由于商业及品牌展示等原因，软件提供商不希望使用者把软件名字隐藏起来。因此，此处需要更改 Nginx 源代码，具体的解决方法如下。



第一步是依次修改 3 个 Nginx 源码文件。

修改的第一个文件为 nginx-1.6.3/src/core/nginx.h, 代码如下:

```
[root@oldboy core]# sed -n '13,17p' nginx.h
#define NGINX_VERSION      "1.6.3"          #<== 修改为想要显示的版本号, 如 2.2.23
#define NGINX_VER          "nginx/" NGINX_VERSION
                                #<== 将 Nginx 修改为想要修改的软件名称, 如 OWS
#define NGINX_VAR          "NGINX"          #<== 将 Nginx 修改为想要修改的软件名称,
                                如 OWS (Oldboy Web Server)
#define NGX_OLDPID_EXT     ".oldbin"
```

修改后的结果如下:

```
[root@oldboy core]# sed -n '13,17p' nginx.h
#define NGINX_VERSION      "2.2.23"          #<== 已修改为想要显示的版本号 2.2.23
#define NGINX_VER          "OWS/" NGINX_VERSION #<== 已修改为想要显示的名字 OWS
#define NGINX_VAR          "OWS"            #<== 已修改为想要显示的名字 OWS
#define NGX_OLDPID_EXT     ".oldbin"
```

修改的第二个文件是 nginx-1.6.3/src/http/nginx\_http\_header\_filter\_module.c 的第 49 行, 需要修改的字符串如下:

```
[root@oldboy http]# grep -n 'Server: nginx' ngx_http_header_filter_module.c
49:static char ngx_http_server_string[] = "Server: nginx " CRLF; #<== 修改本行结尾
    的 Nginx。
```

通过 sed 替换修改, 当然, 也可以通过编辑器进行如下修改:

```
[root@oldboy http]# sed -i 's#Server: nginx#Server: OWS#g' ngx_http_header_
filter_module.c
#<== 把 Server: nginx 替换为 Server: OWS, 注意 Nginx 字符串很多, 所以加上 Server: nginx 一
    起替换。当然也可以指定行替换, 这样就可以直接替换 nginx 字符串了。sed -i '49 s#nginx#OWS#g'
    ngx_http_header_filter_module.c
```

修改后的结果如下:

```
[root@oldboy http]# grep -n 'Server: OWS' ngx_http_header_filter_module.c
49:static char ngx_http_server_string[] = "Server: OWS " CRLF; #<== 变成 OWS 了。
```

修改的第三个文件是 nginx-1.6.3/src/http/nginx\_http\_special\_response.c, 对外页面报错时, 它会控制是否展示敏感信息。这里输出修改前的信息 ngx\_http\_special\_response.c 中的第 21 ~ 30 行, 代码如下:

```
[root@oldboy http]# sed -n '21,30p' ngx_http_special_response.c
static u_char ngx_http_error_full_tail[] =
"<hr><center>" NGINX_VER "</center>" CRLF#<== 此行需要修改
"</body>" CRLF
```

```
"</html>" CRLF
```

```
;
```

```
static u_char ngx_http_error_tail[] =
```

```
"<hr><center>Nginx</center>" CRLF#<== 此行需要修改
```

```
"</body>" CRLF
```

将上述内容中的 ““<hr><center>” NGINX\_VER “</center>” CRLF” 修改为 ““<hr><center>” NGINX\_VER “(http://oldboy.blog.51cto.com)</center>” CRLF”，然后将 ““<hr><center>Nginx </center>” CRLF” 修改为 ““<hr><center>OWS</center>” CRLF”。

修改后的结果如下：

```
[root@oldboy http]# sed -n '21,30p' ngx_http_special_response.c
```

```
static u_char ngx_http_error_full_tail[] =
```

是定义对外展示的内容

```
"<hr><center>" NGINX_VER " (http://oldboy.blog.51cto.com)</center>" CRLF#<== 此行
```

```
"</body>" CRLF
```

```
"</html>" CRLF
```

```
;
```

```
static u_char ngx_http_error_tail[] =
```

```
"<hr><center>OWS</center>" CRLF
```

#<== 此行将对外展示的 Nginx 名字更改为 OWS

```
"</body>" CRLF
```

第二步是修改后编译软件，使其生效。

修改后再编译安装软件，如果是已安装好的服务，则需要重新编译 Nginx，配好配置，启动服务。

再次使浏览器出现 404 错误，然后查看访问结果，如图 12-3 所示：



图 12-3 去掉软件名字和版本的 404 页面图

如图 12-3 所示，Nginx 的软件和版本名都被改掉了，并且加上了我们自己的博客地址。再看看 Linux curl 命令响应的头部信息，代码如下：

```
[root@oldboy http]# curl -I bbs.etiantian.org/oldboy/
```

```
HTTP/1.1 404 Not Found
```

```
Server: OWS/2.2.23
```

#<== 也更改了

```
Date: Thu, 12 Feb 2015 07:00:52 GMT
```

```
Content-Type: text/html
Content-Length: 198
Connection: keep-alive
```



说明

- 提升网站安全，要从最简单、最短板、最低点的地方入手解决问题，如果门开着，即使给窗户安装再结实的护栏也没有意义。
- 向有经验的人及优秀的网站公司学习。
- 学会看官方文档，根据第一手资料去分析。
- 命令输出结果中含有需要过滤或要保留的内容时，命令自身可能有参数可直接实现。
- 掌握技术思想比解决问题本身更重要，详情请见老男孩的博客：<http://oldboy.blog.51cto.com/2561410/1196298>。

### 12.1.3 更改 Nginx 服务的默认用户

为了让 Web 服务更安全，需要尽可能地改掉软件默认的所有配置，包括端口、用户等。

下面就来更改 Nginx 服务的默认用户。

首先，查看 Nginx 服务对应的默认用户。一般情况下，Nginx 服务启动后，默认使用的用户是 nobody，查看默认的配置文件的代码如下：

```
[root@oldboy conf]# grep 'user' nginx.conf.default
#user nobody;
```

为了防止黑客猜到这个 Web 服务的用户，我们需要将其更改成特殊的用户名，例如 nginx 或特殊点的 inca，但是这个用户必须是系统里事先存在的，下面以 nginx 用户为例进行说明。

#### (1) 为 Nginx 服务建立新用户。

为 Nginx 服务建立新用户的操作过程如下：

```
useradd nginx -s /sbin/nologin -M
#<== 不需要有系统登录权限，应当禁止其登录能力，相当于 Apache 里的用户
id nginx #<== 检查用户
```

#### (2) 配置 Nginx 服务，让其使用刚建立的 nginx 用户。

更改 Nginx 服务默认使用的用户，方法有两种：

第一种为直接更改配置文件参数，将默认的“#user nobody;”改为如下内容：

```
user nginx nginx;
```

如果注释或不设置上述参数，则默认为 nobody 用户，不推荐使用 nobody 用户名，最好采用一个普通的用户名，此处用大家习惯的、前文建立好的 nginx 用户。

第二种方法为在编译 Nginx 软件时直接指定编译的用户和组，命令如下：

```
./configure --user=nginx --group=nginx --prefix=/application/nginx1.6.3 --with-http_stub_status_module --with-http_ssl_module
```



**提示** 前文在编译 Nginx 服务时，就是这样带着参数的，因此无论配置文件中是否添加参数，默认都是 nginx 用户。

### (3) 检查更改用户的效果。

重新加载配置后，检查 Nginx 服务进程的对应用户，代码如下：

```
[root@oldboy conf]# ps -ef|grep nginx|grep -v grep
root      1428      1  0 09:56 ?           00:00:00 nginx: master process /
application/nginx/sbin/nginx
nginx     1610    1428  0 10:03 ?           00:00:00 nginx: worker process
nginx     1611    1428  0 10:03 ?           00:00:00 nginx: worker process
```

通过查看上述更改后的 Nginx 进程，可以看到 worker processes 对应的用户都变成了 nginx。所以，我们有理由得出结论，上述的两种方法都可设置 Nginx 的 worker 进程运行的用户。当然，Nginx 的主进程还是以 root 身份运行的，本章后面也会有更改 root 主进程服务用户的深度安全优化与架构技巧讲解。

## 12.2 根据参数优化 Nginx 服务性能

### 12.2.1 优化 Nginx 服务的 worker 进程个数

在高并发、高访问量的 Web 服务场景中，需要事先启动好更多的 Nginx 进程，以保证快速响应并处理大量并发用户的请求。

这类似于开饭店，在营业前，需要事先招聘一定数量的服务员准备接待顾客，但这里存在一个问题，如果饭店对客流量没有一个正确的预估，那么就会导致一些问题发生，例如：服务员招聘过多，客流却很少，那么服务员可能就会很闲，没事干，饭店的成本也高了；如果客流很大，而服务员人数少了，那么可能就会接待不过来顾客，导致顾客吃饭体验差。因此，饭店要根据客户的流量及并发量来调整接待的服务人员数量，然后根据顾客量变化的监测结果及时调整到最佳的配置。

Nginx 服务就相当于饭店，网站用户就相当于顾客，Nginx 的进程就相当于服务员，下面就来讲解如何优化 Nginx 进程的个数。

#### 1. 优化 Nginx 进程对应的配置

优化 Nginx 进程对应 Nginx 服务的配置参数如下：

```
worker_processes 1; #<== 指定了 Nginx 要开启的进程数，结尾的数字就是进程的个数
```

上述参数调整的是 Nginx 服务的 worker 进程数，Nginx 有 Master 进程和 worker 进程之

分, Master 为管理进程, 真正接待“顾客”的是 worker 进程。

## 2. 优化 Nginx 进程个数的策略

前面已经讲解过, worker\_processes 参数大小的设置最好和网站的用户数量相关联, 如果是新配置, 不知道网站的用户数量时该怎么办呢?

搭建服务器时, worker 进程数最开始的设置可以等于 CPU 的核数, 且 worker 进程数要多一些, 这样起始提供服务时就不会出现因为访问量快速增加而需要临时启动新进程提供服务的问题, 缩短了系统的瞬时开销和提供服务的时间, 提升了服务用户的速度。高流量高并发场合也可以考虑将进程数提高至 CPU 核数  $\times 2$ , 具体情况要根据实际的业务来选择, 因为这个参数除了要和 CPU 核数匹配之外, 也与硬盘存储的数据及系统的负载有关, 设置为 CPU 的核数是一个好的起始配置, 这也是官方的建议。

## 3. 查看 Web 服务器 CPU 硬件资源信息

下面介绍查看 Linux 服务器 CPU 总核数的方法。

通过 /proc/cpuinfo 可查看 CPU 的个数及总核数。查看 CPU 总核数的示例如下:

```
[oldboy@oldboy ~]$ grep processor /proc/cpuinfo|wc -l
4 #<== 表示为 1 颗 CPU 四核
[root@oldboy ~]# grep -c processor /proc/cpuinfo
4 #<== 表示为 1 颗 CPU 四核
```

查看 CPU 总颗数的示例如下:

```
[oldboy@oldboy ~]$ grep 'physical id' /proc/cpuinfo|sort|uniq|wc -l
1 #<== 对 physical id 去重计数, 表示 1 颗 CPU
```

通过执行 top 命令, 然后按数字 1, 即可显示所有的 CPU 核数, 代码如下:

```
[root@oldboy-server ~]# top  ← 按 1 显示多核 cpu
top - 11:31:10 up 608 days, 16:04, 2 users, load average: 0.00, 0.00, 0.00
Tasks: 121 total, 1 running, 114 sleeping, 6 stopped, 0 zombie
Cpu0  :  0.2%us,  0.1%sy,  0.0%ni, 99.2%id,  0.5%wa,  0.0%hi,  0.0%si,  0.0%st
Cpu1  :  0.1%us,  0.0%sy,  0.0%ni, 99.5%id,  0.3%wa,  0.0%hi,  0.0%si,  0.0%st
Cpu2  :  0.2%us,  0.1%sy,  0.0%ni, 99.4%id,  0.3%wa,  0.0%hi,  0.0%si,  0.0%st
Cpu3  :  0.2%us,  0.1%sy,  0.0%ni, 99.4%id,  0.3%wa,  0.0%hi,  0.0%si,  0.0%st
Mem:   8173172k total, 8126340k used, 46832k free, 419508k buffers
Swap:  4192956k total, 156k used, 4192800k free, 6681084k cached

  PID USER      PR  NI  VIRT  RES  SHR  S  %CPU  %MEM    TIME+  COMMAND
    1 root        15   0 10368   680  572  S   0.0   0.0   0:01.94 init
    2 root        RT  -5    0    0    0  S   0.0   0.0   0:02.48 migration/0
#<== 这是单 CPU 四核的信息
```

例如: CPU 核数为 4, 就配置 worker\_processes 4

## 4. 实践修改 Nginx 配置

假设服务器的 CPU 颗数为 1 颗, 核数为 4 核, 则初始的配置可通过查看默认的 nginx.

conf 里的 worker\_processes 数来了解，命令如下：

```
[root@oldboy conf]# grep worker_processes nginx.conf
worker_processes 1;
```

这里修改参数值为 CPU 的总核数 4，然后重新加载 Nginx 服务。

修改配置的方法如下：

```
[root@oldboy conf]# sed -i 's#worker_processes 1#worker_processes 4#g' nginx.conf
[root@oldboy conf]# grep worker_processes nginx.conf
worker_processes 4;
```

提示：可以通过 vi 修改

优雅重启 Nginx，使修改生效，代码如下：

```
[root@oldboy ~]# /application/nginx/sbin/nginx -t
nginx: the configuration file /application/nginx1.6.3/conf/nginx.conf syntax is ok
nginx: configuration file /application/nginx1.6.3/conf/nginx.conf test is successful
[root@oldboy ~]# /application/nginx/sbin/nginx -s reload
```

现在检查修改后的 worker 进程数量，代码如下：

```
[root@oldboy ~]# ps -ef|grep nginx|grep -v grep
root      1428      1  0 09:56 ?        00:00:00 nginx: master process /application/
nginx/sbin/nginx
nginx     1832    1428  0 10:24 ?        00:00:00 nginx: worker process
nginx     1833    1428  0 10:24 ?        00:00:00 nginx: worker process
nginx     1834    1428  0 10:24 ?        00:00:00 nginx: worker process
nginx     1835    1428  0 10:24 ?        00:00:00 nginx: worker process
```

从 “worker\_processes 4” 可知，worker 的进程数为 4 个。Nginx Master 主进程不包含在这个参数内，Nginx Master 的主进程为管理进程，负责调度和管理 worker 进程。

有关 worker\_processes 参数的官方说明如下：

```
syntax:    worker_processes number; #<== 此行为参数语法，number 为数量
default:   worker_processes 1;      #<== 此行的意思是不配置该参数，软件默认情况下数量为 1
context:   main                     #<== 此行为 worker_processes 参数可以放置的位置

worker_processes 为定义 worker 进程数的数量，建议设置为 CPU 的核数或 CPU 核数 × 2，具体情况要根据实际的业务来选择，因为这个参数，除了要和 CPU 核数匹配之外，还与硬盘存储的数据及系统的负载有关，设置为 CPU 的个数或核数是一个好的起始配置。
From : http://nginx.org/en/docs/nginx_core_module.html
```

## 12.2.2 优化绑定不同的 Nginx 进程到不同的 CPU 上

默认情况下，Nginx 的多个进程有可能运行在某一个 CPU 或 CPU 的某一核上，导致 Nginx 进程使用硬件的资源不均，本节的优化将尽可能地不同的 Nginx 进程分配给不同的 CPU 处理，达到充分有效利用硬件的多 CPU 多核资源的目的。

在优化不同的 Nginx 进程对应不同的 CPU 配置时，四核 CPU 服务器的参数配置参考



如下:

```
worker_processes 4;
worker_cpu_affinity 0001 0010 0100 1000;
#<== worker_cpu_affinity 就是配置 Nginx 进程与 CPU 亲和力的参数, 即把不同的进程分给不同的 CPU 处理。这里 0001 0010 0100 1000 是掩码, 分别代表第 1、2、3、4 核 CPU, 由于 worker_processes 进程数为 4, 因此, 上述配置会为每个进程分配一核 CPU 处理, 默认情况下进程不会绑定任何 CPU, 参数位置为 main 段。
```

八核 CPU 服务器的参数配置参考如下:

```
worker_cpu_affinity 00000001 00000010 00000100 00001000 00010000 00100000 01000000 10000000;
worker_cpu_affinity 0001 0010 0100 1000 0001 0010 0100 1000;
```

worker\_cpu\_affinity 参数的官方说明如下:

```
syntax: worker_cpu_affinity cpumask ...; # <== 此行为 CPU 亲和力参数语法,
                                           cpumask 为 CPU 掩码
default: — # <== 默认不设置
context: main # <== 此行为 worker_cpu_affinity
                                           参数可以放置的位置
```

worker\_cpu\_affinity 的作用是绑定不同的 worker 进程数到一组 CPU 上。通过设置 bitmask 控制进程允许使用的 CPU, 默认 worker 进程不会绑定到任何 CPU 上。

下面是绑定的示例配置:

```
worker_processes 4;
worker_cpu_affinity 0001 0010 0100 1000;
binds each worker process to a separate CPU, while
worker_processes 2;
worker_cpu_affinity 0101 1010;
binds the first worker process to CPU0/CPU2, and the second worker process to CPU1/CPU3. The second example is suitable for hyper-threading. The directive is only available on FreeBSD and Linux.
From : http://nginx.org/en/docs/nginx\_core\_module.html by oldboy
```

下面是压力测试配置结果。配置前的压力测试结果为:

```
top - 16:30:17 up 41 min, 2 users, load average: 2.30, 1.50, 1.35
Tasks: 104 total, 4 running, 100 sleeping, 0 stopped, 0 zombie
Cpu0 : 6.7%us, 9.3%sy, 0.0%ni, 4.3%id, 0.0%wa, 5.0%hi, 74.7%si, 0.0%st
Cpu1 : 18.3%us, 26.6%sy, 0.0%ni, 48.8%id, 0.0%wa, 0.0%hi, 6.3%si, 0.0%st
Cpu2 : 15.9%us, 24.9%sy, 0.0%ni, 54.8%id, 0.0%wa, 0.0%hi, 4.3%si, 0.0%st
Cpu3 : 16.0%us, 27.0%sy, 0.0%ni, 53.3%id, 0.0%wa, 0.0%hi, 3.7%si, 0.0%st

top - 16:30:38 up 41 min, 2 users, load average: 2.65, 1.63, 1.39
Tasks: 104 total, 3 running, 101 sleeping, 0 stopped, 0 zombie
Cpu0 : 7.7%us, 11.3%sy, 0.0%ni, 4.7%id, 0.0%wa, 5.0%hi, 71.3%si, 0.0%st
Cpu1 : 17.3%us, 29.2%sy, 0.0%ni, 47.8%id, 0.3%wa, 0.0%hi, 5.3%si, 0.0%st
```



```
Cpu2 : 17.7%us, 24.0%sy, 0.0%ni, 53.7%id, 0.0%wa, 0.0%hi, 4.7%si, 0.0%st
Cpu3 : 17.7%us, 27.7%sy, 0.0%ni, 51.3%id, 0.3%wa, 0.0%hi, 3.0%si, 0.0%st
```

```
top - 16:30:57 up 42 min, 2 users, load average: 2.48, 1.65, 1.41
Tasks: 104 total, 5 running, 99 sleeping, 0 stopped, 0 zombie
Cpu0 : 6.0%us, 11.4%sy, 0.0%ni, 4.7%id, 0.0%wa, 5.0%hi, 72.9%si, 0.0%st
Cpu1 : 17.9%us, 26.9%sy, 0.0%ni, 50.5%id, 0.0%wa, 0.0%hi, 4.7%si, 0.0%st
Cpu2 : 16.9%us, 26.8%sy, 0.0%ni, 51.7%id, 0.0%wa, 0.0%hi, 4.6%si, 0.0%st
Cpu3 : 16.6%us, 27.6%sy, 0.0%ni, 51.8%id, 0.0%wa, 0.0%hi, 4.0%si, 0.0%st
```

```
top - 16:31:22 up 42 min, 2 users, load average: 2.65, 1.76, 1.45
Tasks: 104 total, 4 running, 100 sleeping, 0 stopped, 0 zombie
Cpu0 : 7.7%us, 12.7%sy, 0.0%ni, 5.3%id, 0.0%wa, 5.0%hi, 69.3%si, 0.0%st
Cpu1 : 19.4%us, 28.4%sy, 0.0%ni, 47.2%id, 0.0%wa, 0.0%hi, 5.0%si, 0.0%st
Cpu2 : 17.0%us, 26.3%sy, 0.0%ni, 52.3%id, 0.0%wa, 0.0%hi, 4.3%si, 0.0%st
Cpu3 : 16.9%us, 25.6%sy, 0.0%ni, 54.2%id, 0.0%wa, 0.0%hi, 3.3%si, 0.0%st
```

通过观察，我们发现配置前不同 CPU 的使用率相对平均。

配置 `worker_cpu_affinity`，代码如下：

```
[root@www ~]# grep worker_cpu nginx.conf
worker_cpu_affinity 0001 0010 0100 1000;
```

压力测试的命令及结果如下：

```
[root@www ~]# webbench -c 20000 -t 180 http://10.0.0.190/
Webbench - Simple Web Benchmark 1.5
Copyright (c) Radim Kolar 1997-2004, GPL Open Source Software.
Benchmarking: GET http://10.0.0.190/
20000 clients, running 180 sec.
Speed=1521283 pages/min, 3588193 bytes/sec.
Requests: 3661785 succeed, 902066 failed.
```

查看 CPU 调度的结果如下：

```
top - 16:24:58 up 36 min, 2 users, load average: 1.65, 1.58, 1.34
Tasks: 104 total, 4 running, 100 sleeping, 0 stopped, 0 zombie
Cpu0 : 4.7%us, 8.0%sy, 0.0%ni, 7.3%id, 0.0%wa, 5.3%hi, 74.8%si, 0.0%st
Cpu1 : 16.9%us, 32.2%sy, 0.0%ni, 43.2%id, 0.0%wa, 0.0%hi, 7.6%si, 0.0%st
Cpu2 : 16.6%us, 26.2%sy, 0.0%ni, 52.2%id, 0.0%wa, 0.0%hi, 5.0%si, 0.0%st
Cpu3 : 16.9%us, 28.2%sy, 0.0%ni, 49.8%id, 0.0%wa, 0.0%hi, 5.0%si, 0.0%st
top - 16:25:36 up 36 min, 2 users, load average: 1.83, 1.64, 1.37
Tasks: 104 total, 2 running, 102 sleeping, 0 stopped, 0 zombie
Cpu0 : 5.0%us, 8.1%sy, 0.0%ni, 9.7%id, 0.0%wa, 5.0%hi, 72.1%si, 0.0%st
Cpu1 : 17.6%us, 31.2%sy, 0.0%ni, 43.9%id, 0.0%wa, 0.0%hi, 7.3%si, 0.0%st
Cpu2 : 16.0%us, 26.7%sy, 0.0%ni, 52.7%id, 0.0%wa, 0.0%hi, 4.7%si, 0.0%st
Cpu3 : 17.9%us, 28.6%sy, 0.0%ni, 48.8%id, 0.0%wa, 0.0%hi, 4.7%si, 0.0%st
```

```
top - 16:26:09 up 37 min, 2 users, load average: 2.01, 1.71, 1.40
Tasks: 104 total, 4 running, 100 sleeping, 0 stopped, 0 zombie
```

```
Cpu0 : 6.3%us, 9.0%sy, 0.0%ni, 10.6%id, 0.0%wa, 4.7%hi, 69.4%si, 0.0%st
Cpu1 : 19.3%us, 32.6%sy, 0.0%ni, 41.9%id, 0.0%wa, 0.0%hi, 6.3%si, 0.0%st
Cpu2 : 17.9%us, 26.9%sy, 0.0%ni, 51.8%id, 0.0%wa, 0.0%hi, 3.3%si, 0.0%st
Cpu3 : 19.7%us, 30.0%sy, 0.0%ni, 46.3%id, 0.0%wa, 0.0%hi, 4.0%si, 0.0%st
```

```
top - 16:26:22 up 37 min, 2 users, load average: 2.21, 1.77, 1.42
Tasks: 104 total, 4 running, 100 sleeping, 0 stopped, 0 zombie
Cpu0 : 5.3%us, 8.3%sy, 0.0%ni, 10.3%id, 0.0%wa, 5.0%hi, 71.0%si, 0.0%st
Cpu1 : 19.3%us, 29.2%sy, 0.0%ni, 45.8%id, 0.0%wa, 0.0%hi, 5.6%si, 0.0%st
Cpu2 : 15.9%us, 24.3%sy, 0.0%ni, 55.8%id, 0.0%wa, 0.0%hi, 4.0%si, 0.0%st
Cpu3 : 16.7%us, 27.0%sy, 0.0%ni, 52.7%id, 0.0%wa, 0.0%hi, 3.7%si, 0.0%st
```

通过观察,我们发现配置后不同 CPU 的使用率相对平均,和测试前的变化不大。因此可知,默认就是比较平均的,一方面,可能是 Nginx 软件自身正在逐渐优化,使其使用多核 CPU 时更为均衡;另一方面,测试的数据可能还有待调整。

另外 (taskset - retrieve or set a process' s CPU affinity) 命令本身也有分配 CPU 的功能,这里留给大家测试 (例如: taskset -c 1,2,3 /etc/init.d/mysql start):

```
-c, --cpu-list
        specify a numerical list of processors instead of a bitmask.
        The list may contain multiple items, separated by comma, and
        ranges. For example, 0,5,7,9-11.
```

### 12.2.3 Nginx 事件处理模型优化

Nginx 的连接处理机制在不同的操作系统中会采用不同的 I/O 模型,在 Linux 下, Nginx 使用 epoll 的 I/O 多路复用模型,在 FreeBSD 中使用 kqueue 的 I/O 多路复用模型,在 Solaris 中使用 /dev/poll 方式的 I/O 多路复用模型,在 Windows 中使用的是 icop, 等等。

要根据系统类型选择不同的事件处理模型,可供使用的选择有 “use [ kqueue | rtsig | epoll | /dev/poll | select | poll ];”。本书使用的是 CentOS 6.6 Linux, 因此将 Nginx 的事件处理模型调整为 epoll 模型。

具体的配置参数如下:

```
events
#<==events 指令是设定 Nginx 的工作模式及连接数上限
{
    use epoll;
#<==use 是一个事件模块指令,用来指定 Nginx 的工作模式。Nginx 支持的工作模式有 select、poll、
kqueue、epoll、rtsig 和 /dev/poll。其中 select 和 poll 都是标准的工作模式,kqueue 和 epoll
是高效的工作模式,不同的是 epoll 用在 Linux 平台上,而 kqueue 用在 BSD 系统中。对于 Linux 系统
Linux 2.6+ 的内核,推荐选择 epoll 工作模式,这是高性能高并发的设置
}
```

根据 Nginx 的官方文档建议,也可以不指定事件处理模型, Nginx 会自动选择最佳的事件处理模型服务。

events 区块及 use 事件处理参数的官方说明如下：

```
syntax:    events { ... }      # <== 语法配置
default:   —                  # <== 默认没有设置
context:   main               # <==events 标签的放置位置，放在 main 段
```

events 区块是一个用来设置连接进程的区块，例如：设置 Nginx 的网络 I/O 模型，以及连接数等。

use 事件的处理参数说明如下：

```
syntax:    use method;        # <== 网络模型配置，method 选择模型之一
default:   —                  # <== 默认没有设置
context:   events             # <== 网络模型配置放置于 events 区块内
```

对于使用连接进程的方法，通常不需要进行任何设置，Nginx 会自动选择最有效的方法。



**提示** 事件处理模型参数需要在 events 区块中进行配置。

## 12.2.4 调整 Nginx 单个进程允许的客户端最大连接数

接下来，调整 Nginx 单个进程允许的客户端最大连接数，控制连接数的参数为 worker\_connections。

worker\_connections 的值要根据具体服务器的性能和程序的内存使用量来指定（单个进程启动使用的内存根据程序来确定），代码如下：

```
events #<==events 指令是设定 Nginx 的工作模式及连接数上限
{
    worker_connections 20480;
    #<==worker_connections 也是个事件模块指令，用于定义 Nginx 每个进程的最大连接数，默认是 1024。
    最大客户端连接数由 worker_processes 和 worker_connections 决定，即 Max_client= worker_
    processes*worker_connections。 进程的最大连接数受 Linux 系统进程的最大打开文件数限制，在执行
    操作系统命令“ulimit -HSn 65535”或配置相应文件后，worker_connections 的设置才能生效
```

下面是 worker\_connections 的官方说明。

参数语法：worker\_connections number

默认配置：worker\_connections 512

放置位置：events

说明：worker\_connections 用来设置一个 worker process 支持的最大并发连接数，这个连接数包括了所有连接，例如：代理服务器的连接、客户端的连接等，实际的并发连接数除了受 worker\_connections 参数控制之外，还与最大打开文件数 worker\_rlimit\_nofile 有关（见下文），Nginx 总并发连接 = worker 数量 × worker\_connections。

参考资料：[http://nginx.org/en/docs/nginx\\_core\\_module.html](http://nginx.org/en/docs/nginx_core_module.html)。

### 12.2.5 配置 Nginx worker 进程的最大打开文件数

接下来，调整配置 Nginx worker 进程的最大打开文件数，这个控制连接数的参数为 `worker_rlimit_nofile`。该参数的实际配置如下：

```
worker_rlimit_nofile 65535;
```

#<== 最大打开文件数，可设置为系统优化后的 `ulimit -HSn` 的结果

下面是 `worker_rlimit_nofile` 的官方说明。

参数语法：`worker_rlimit_nofile number`

默认配置：无

放置位置：主标签段

说明：此参数的作用是改变 worker processes 能打开的最大文件数

参考资料：[http://nginx.org/en/docs/nginx\\_core\\_module.html](http://nginx.org/en/docs/nginx_core_module.html)

### 12.2.6 优化服务器域名的散列表大小

先将确切名字和通配符名字存储在散列表中。散列表与监听端口关联，每个端口最多关联到三张表：确切名字的散列表、以星号起始的通配符名字的散列表和以星号结束的通配符名字的散列表。散列表的尺寸在配置阶段进行了优化，可以以最小的 CPU 缓存命中失败来找到名字。Nginx 首先会搜索确切名字的散列表；如果没有找到，则搜索以星号起始的通配符名字的散列表；如果还是没有找到，则继续搜索以星号结束的通配符名字的散列表。因为名字是按照域名的字节来搜索的，所以搜索通配符名字的散列表比搜索确切名字的散列表慢。注意 `.nginx.org` 存储在通配符名字的散列表中，而不在确切名字的散列表中。由于正则表达式是逐个进行串行测试的，因此该方式也是最慢的，而且不可扩展。

鉴于以上原因，请尽可能地使用确切的名称。举个例子，如果使用 `nginx.org` 和 `www.nginx.org` 来访问服务器是最频繁的，那么将它们明确地定义出来就更为有效，命令如下：

```
server {
    listen      80;
    server_name nginx.org www.nginx.org *.nginx.org;
    ...
}
```

下面这种方法更简单，但是效率也更低：

```
server {
    listen      80;
    server_name .nginx.org;
    ...
}
```

如果定义了大量的名字，或者定义了非常长的名字，那就需要在 HTTP 配置块中调整

`server_names_hash_max_size` 和 `server_names_hash_bucket_size` 的值。`server_names_hash_bucket_size` 的默认值可能是 32 或 64，也可能是其他值，这取决于 CPU 的缓存行的长度。如果这个值是 32，那么定义 “`too.long.server.name.nginx.org`” 作为虚拟主机名就会失败，此时会显示如下的错误信息：

```
could not build the server_names_hash,
you should increase server_names_hash_bucket_size: 32
```

若出现了这种情况，那就需要将设置值扩大一倍，命令如下：

```
http {
    server_names_hash_bucket_size 64;
    ...
}
```

如果定义了大量名字，则会得到如下的另外一个错误信息：

```
could not build the server_names_hash,
you should increase either server_names_hash_max_size: 512
or server_names_hash_bucket_size: 32
```

应该先尝试设置 `server_names_hash_max_size` 的值，此值差不多等于名字列表的名字总量。如果还不能解决问题，或者服务器启动非常缓慢，那么再尝试增加 `server_names_hash_bucket_size` 的值，具体信息如下：

```
server_names_hash_max_size 512;           # 默认是 512KB，一般要查看系统给出确切
                                           的值。这里一般是 CPU L1 的 4-5 倍
```

`server_names_hash_max_size` 的官方说明如下：

```
syntax:    server_names_hash_max_size size;           #<== 参数语法
default:    server_names_hash_max_size 512;           #<== 参数默认大小
context:    http #<== 仅能放置在 http 标签段
```

参数作用：设置存放域名（`server names`）的最大散列表大小。细节见 [http://nginx.org/en/docs/nginx\\_core\\_module.html](http://nginx.org/en/docs/nginx_core_module.html)。

第二个参数如下：

```
server_names_hash_bucket_size 128;
# <== 不能带单位！配置主机时必须设置该值，否则无法运行 Nginx，或者无法通过测试。该设置与 server_names_hash_max_size 共同控制保存服务器名的 hash 表，hash bucket size 总是等于 hash 表的大小，并且是一路处理器缓存大小的倍数。若 hash bucket size 等于一路处理器缓存的大小，那么在查找键时，最坏的情况下在内存中查找的次数为 2。第一次是确定存储单元的地址，第二次是在存储单元中查找键值。若报出 hash max size 或 hash bucket size 的提示，则需要增加 server_names_hash_max_size 的值
```

`server_names_hash_bucket_size` 的官方说明如下：

```
syntax:    server_names_hash_bucket_size size;         #<== 参数语法
default:    server_names_hash_bucket_size 32|64 |128;   #<== 参数默认大小
context:    http#<== 仅能放置在 http 标签段
```

参数作用：设置存放域名（server names）的最大散列表的存储桶（bucket）的大小。默认值依赖 CPU 的缓存行。细节见 [http://nginx.org/en/docs/nginx\\_core\\_module.html](http://nginx.org/en/docs/nginx_core_module.html)。

## 12.2.7 开启高效文件传输模式

### 1. 设置参数：sendfile on;

sendfile 参数用于开启文件的高效传输模式。同时将 tcp\_nopush 和 tcp\_nodelay 两个指令设置为 on，可防止网络及磁盘 I/O 阻塞，提升 Nginx 工作效率。

sendfile 参数的官方说明如下：

syntax:	sendfile on   off;	#<== 参数语法
default:	sendfile off;	#<== 参数默认大小
context:	http, server, location, if in location	#<== 可以放置的标签段

参数作用：激活或禁用 sendfile() 功能。sendfile() 是作用于两个文件描述符之间的数据复制函数，这个复制操作是发生在内核之中的，被称为“零复制”，sendfile() 比 read 和 write 函数要高效很多，因为，read 和 write 函数要把数据复制到应用层再进行操作。相关控制参数还有 sendfile\_max\_chunk，读者可以自行查询。细节见 [http://nginx.org/en/docs/http/nginx\\_core\\_module.html#sendfile](http://nginx.org/en/docs/http/nginx_core_module.html#sendfile)。

### 2. 设置参数：tcp\_nopush on;

tcp\_nopush 参数的官方说明如下：

syntax:	tcp_nopush on   off;	#<== 参数语法
default:	tcp_nopush off;	#<== 参数默认大小
context:	http, server, location	#<== 可以放置的标签段

参数作用：激活或禁用 Linux 上的 TCP\_CORK socket 选项，此选项仅仅当开启 sendfile 时才生效，激活这个 tcp\_nopush 参数可以允许把 http response header 和文件的开始部分放在一个文件里发布，其积极的作用是减少网络报文段的数量。细节见 [http://nginx.org/en/docs/http/nginx\\_http\\_core\\_module.html](http://nginx.org/en/docs/http/nginx_http_core_module.html)。

## 12.2.8 优化 Nginx 连接参数，调整连接超时时间

### 1. 什么是连接超时

先来个比喻吧，某饭店请了服务员招待顾客，但是现在饭店不景气，此时，为多余的服务员发工资使得成本被提高，想减少饭店开支成本就得解雇服务员。

这里的服务员就相当于 Nginx 服务建立的连接，当服务器建立的连接没有接收处理请求时，可在指定的时间内让它超时自动退出。还有当 Nginx 和 FastCGI 服务建立连接请求 PHP 时，如果因为一些原因（负载高、停止响应），FastCGI 服务无法给 Nginx 返回数据，此时可以通过配置 Nginx 服务参数使其不会死等，因为前面的用户还等着它返回数据呢，例如，可设置为如果请求 FastCGI 10 秒内不能返回数据，那么 Nginx 就中断本次请求，向用户汇报取



不到数据的错误。

## 2. 连接超时的作用

- ❑ 将无用的连接设置为尽快超时，可以保护服务器的系统资源（CPU、内存、磁盘）。
- ❑ 当连接很多时，及时断掉那些已经建立好的但又长时间不做事的连接，以减少其占用的服务器资源，因为服务器维护连接也是消耗资源的。
- ❑ 有时黑客或恶意用户攻击网站，就会不断地和服务器建立多个连接，消耗连接数，但是啥也不干，大量消耗服务器的资源，此时就应该及时断掉这些恶意占用资源的连接。
- ❑ LNMP 环境中，如果用户请求了动态服务，则 Nginx 就会建立连接，请求 FastCGI 服务及后端 MySQL 服务，此时这个 Nginx 连接就要设定一个超时时间，在用户容忍的时间内返回数据，或者再多等一会后端服务返回数据，具体的策略要根据具体业务进行分析。当然了，后端的 FastCGI 服务及 MySQL 服务也有对连接的超时控制。

简单地说，连接超时是服务自我管理、自我保护的一种重要机制。

## 3. 连接超时带来的问题，以及不同程序连接设定的知识

服务器建立新连接也是要消耗资源的，因此，超时设置得太短而并发很大，就会导致服务器瞬间无法响应用户的请求，导致用户体验下降。

企业生产中有些 PHP 程序站点会希望设置成短连接，因为 PHP 程序建立连接消耗的资源和时间相对要少些。而对于 Java 程序站点来说，一般建议设置成长连接，因为 Java 程序建立连接消耗的资源和时间更多，这是语言运行机制决定的。

## 4. Nginx 连接超时的参数设置

### 1) 设置参数：keepalive\_timeout 60;

用于设置客户端连接保持会话的超时时间为 60 秒。若超过这个时间，服务器就会关闭该连接，此数值仅为参考值。

keepalive\_timeout 参数的官方说明如下：

```
syntax:    keepalive_timeout timeout [header_timeout];  #<== 参数语法
default:   keepalive_timeout 75s;                      #<== 参数默认大小
context:   http, server, location                      #<== 可以放置的标签段
```

参数作用：keep-alive 可以使客户端到服务器端已经建立的连接一直工作不退出，当服务器有持续请求时，keep-alive 会使用已经建立的连接提供服务，从而避免服务器重新建立新连接处理请求。

此参数设置一个 keep-alive（客户端连接在服务器端保持多久后退出），其单位是秒，和 HTTP 响应 header 域的“Keep-Alive: timeout=time”参数有关，这些 header 信息也会被客户端浏览器识别并处理，不过有些客户端并不能按照服务器端的设置来处理，例如：MSIE 大约 60 秒后会关闭 keep-alive 连接。细节见 [http://nginx.org/en/docs/http/nginx\\_http\\_core\\_module.html](http://nginx.org/en/docs/http/nginx_http_core_module.html)。



## 2) 设置参数: `tcp_nodelay on;`

用于激活 `tcp_nodelay` 功能, 提高 I/O 性能。

`tcp_nodelay` 参数的官方说明如下:

```
syntax:      tcp_nodelay on | off;           #<== 参数语法
default:     tcp_nodelay on;                 #<== 参数默认大小
context:     http, server, location          #<== 可以放置的标签段
```

参数作用: 默认情况下当数据发送时, 内核并不会马上发送, 可能会等待更多的字节组成一个数据包, 这样可以提高 I/O 性能。但是, 在每次只发送很少字节的业务场景中, 使用 `tcp_nodelay` 功能, 等待时间会比较长。

参数生效条件: 激活或禁用 `TCP_NODELAY` 选项, 当一个连接进入 `keep-alive` 状态时生效。细节见 [http://nginx.org/en/docs/http/nginx\\_http\\_core\\_module.html](http://nginx.org/en/docs/http/nginx_http_core_module.html)。

## 3) 设置参数: `client_header_timeout 15;`

用于设置读取客户端请求头数据的超时时间。此处的数值为 15, 其单位是秒, 为经验参考值。

`client_header_timeout` 参数的官方说明如下:

```
syntax:      client_header_timeout time;     #<== 参数语法
default:     client_header_timeout 60s;      #<== 参数默认大小
context:     http, server                     #<== 可以放置的标签段
```

参数作用: 设置读取客户端请求头数据的超时时间。如果超过这个时间, 客户端还没有发送完整的 `header` 数据, 那么服务器端将返回 “Request time out (408)” 错误, 可指定一个超时时间, 防止客户端利用 `http` 协议进行攻击。细节见 [http://nginx.org/en/docs/http/nginx\\_http\\_core\\_module.html](http://nginx.org/en/docs/http/nginx_http_core_module.html)。

## 4) 设置参数: `client_body_timeout 15;`

用于设置读取客户端请求主体的超时时间, 默认值是 60。

`client_body_timeout` 参数的官方说明如下:

```
syntax:      client_body_timeout time;       #<== 参数语法
default:     client_body_timeout 60s;        #<== 默认值是 60 秒
context:     http, server, location           #<== 可以放置的标签段
```

参数作用: 设置读取客户端请求主体的超时时间。这个超时仅仅为两次成功读取操作之间的一个超时, 而不是请求整个主体数据的超时时间, 如果在这个超时时间内, 客户端没有发送任何数据, Nginx 将返回 “Request time out (408)” 错误, 默认值是 60, 这样效果更好。细节见 [http://nginx.org/en/docs/http/nginx\\_http\\_core\\_module.html](http://nginx.org/en/docs/http/nginx_http_core_module.html)。

## 5) 设置参数: `send_timeout 25;`

用于指定响应客户端的超时时间。这个超时仅限于两个连接活动之间的时间, 如果超过这个时间, 客户端没有任何活动, Nginx 将会关闭连接, 默认值为 60 秒, 可以改为参考值

25 秒。

send\_timeout 参数的官方说明如下：

```
syntax:      send_timeout time;           #<== 参数语法
default:    send_timeout 60s;           #<== 默认值是 60 秒
context:    http, server, location      #<== 可以放置的标签段
```

参数作用：设置服务器端传送 HTTP 响应信息到客户端的超时时间，这个超时仅仅为两次成功握手后的一个超时，而不是请求整个响应数据的超时时间，如在这个超时时间内，客户端没有接收任何数据，那么连接将被关闭。细节见 [http://nginx.org/en/docs/http/nginx\\_http\\_core\\_module.html](http://nginx.org/en/docs/http/nginx_http_core_module.html)。

下面画图（如图 12-4 所示）讲解上述几个超时参数。

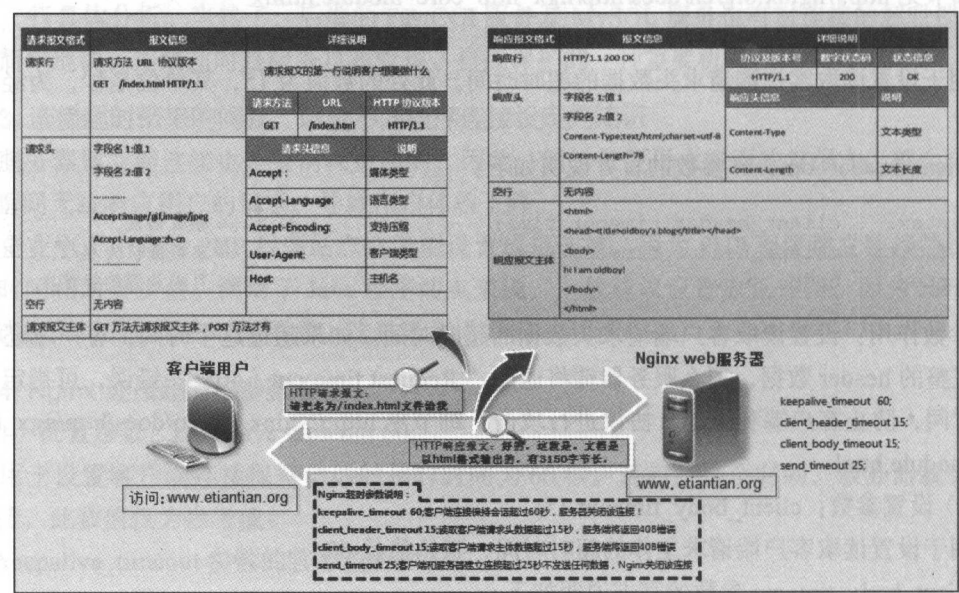


图 12-4 画图讲解 Nginx 配置超时参数图

### 12.2.9 上传文件大小的限制（动态应用）

下面介绍如何调整上传文件的大小（http Request body size）限制。

首先，在 Nginx 的主配置文件里加入如下参数：

```
client_max_body_size 8m;
```

具体大小可根据公司的业务做调整，如果不清楚就先设置为 8m 吧，一般情况下，HTTP 的 post 方法在提交数据时才会携带请求主体信息。

client\_max\_body\_size 参数的官方说明如下：

```

syntax:    client_max_body_size size;#<== 参数语法
default:   client_max_body_size 1m;  #<== 默认值是 1m
context:   http, server, location    #<== 可以放置的标签段

```

参数作用：设置允许的最大客户端请求主体大小，在请求头域有“Content-Length”，如果超过了此配置值，则客户端会收到 413 错误，意思是请求的条目过大，有可能浏览器不能正确显示。设置为 0 则表示禁止检查客户端请求主体大小。此参数对提高服务器端的安全性有一定的作用。细节见 [http://nginx.org/en/docs/http/nginx\\_http\\_core\\_module.html](http://nginx.org/en/docs/http/nginx_http_core_module.html)。

### 12.2.10 FastCGI 相关参数调优（配合 PHP 引擎动态服务）

FastCGI 参数是配合 Nginx 向后请求 PHP 动态引擎服务的相关参数，要想较好地理解参数细节，需要熟悉和了解 Nginx FastCGI 客户端配合 php-fpm（FastCGI 服务器端）的原理。Nginx FastCGI 工作的逻辑图如图 12-5 所示。

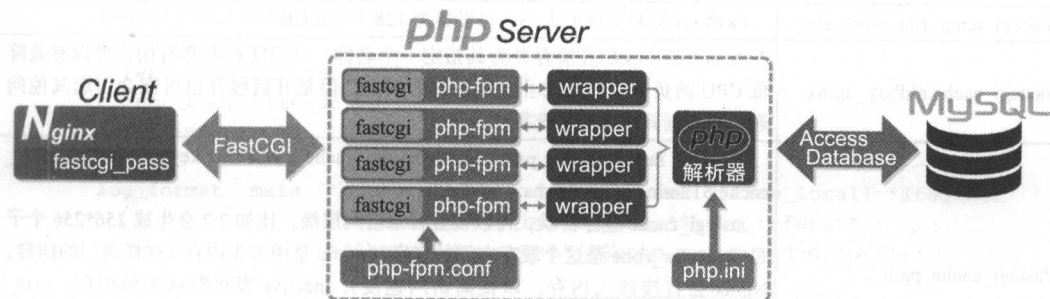


图 12-5 Nginx FastCGI 模式工作原理图

此处讲解的参数均为 Nginx FastCGI 客户端向后请求 PHP 动态引擎服务（php-fpm（FastCGI 服务器端））的相关参数，属于 Nginx 的配置参数。表 12-1 是 Nginx FastCGI 常见参数的说明。

表 12-1 Nginx FastCGI 常见参数列表说明

Nginx FastCGI 相关参数	说明
fastcgi_connect_timeout	表示 Nginx 服务器和后端 FastCGI 服务器连接的超时时间，默认值为 60 秒，这个参数值通常不要超过 75 秒，因为建立的连接越多，消耗的资源就越多
fastcgi_send_timeout	设置 Nginx 允许 FastCGI 服务器端返回数据的超时时间，即在规定时间内后端服务器必须传完所有的数据，否则，Nginx 将断开这个连接。其默认值为 60 秒
fastcgi_read_timeout	设置 Nginx 从 FastCGI 服务器端读取响应信息的超时时间，表示连接建立成功后，Nginx 等待后端服务器的响应时间，是 Nginx 已经进入后端的排队之中等候处理的时间
fastcgi_buffer_size	这是 Nginx FastCGI 的缓冲区大小参数，设定用来读取从 FastCGI 服务器端收到的第一部分响应信息的缓冲区大小，这里的第一部分通常会包含一个小的响应头部。默认情况下，这个参数的大小是由 fastcgi_buffers 指定的一个缓冲区的大小

(续)

Nginx FastCGI 相关参数	说明
fastcgi_buffers	<p>设定用来读取从 FastCGI 服务器端收到的响应信息的缓冲区大小和缓冲区数量，默认值为 fastcgi_buffers 8 4k 8k;。</p> <p>指定本地需要用多少和多大的缓冲区来缓冲 FastCGI 的应答请求。如果一个 PHP 脚本所产生的页面大小为 256KB，那么会为其分配 4 个 64KB 的缓冲区来缓存；如果页面大小大于 256KB，那么大于 256KB 的部分会缓存到 fastcgi_temp 指定的路径中，但是这并不是好的方法，因为内存中的数据处理速度要快于硬盘。一般这个值应该为站点中 PHP 脚本所产生的页面大小的中间值，如果站点大部分脚本所产生的页面大小为 256KB，那么可以把这个值设置为“16 16k”、“4 64k”等</p>
proxy_busy_buffers_size	用于设置系统很忙时可以使用 proxy_buffers 大小，官方推荐的大小为 proxy_buffers*2
fastcgi_busy_buffers_size	<p>用于设置系统很忙时可以使用 fastcgi_buffers 大小，官方推荐的大小为 fastcgi_buffers *2。</p> <p>默认值为 fastcgi_busy_buffers_size 8k 16k</p>
fastcgi_temp_file_write_size	FastCGI 临时文件的大小，可设置为 128 ~ 256KB
fastcgi_cache oldboy_nginx	表示开启 FastCGI 缓存并为其指定一个名称。开启缓存非常有用，可以有效降低 CPU 的负载，并且防止 502 错误的发生，但是开启缓存也可能引起其他问题，要根据具体情况来选择
fastcgi_cache_path	<p>示例：fastcgi_cache_path /data/nginx_fcgi_cache levels=2:2 keys_zone= ngx_fcgi_cache:512m inactive=1d max_size=40g;</p> <p>fastcgi_cache 缓存目录，可以设置目录前列层级，比如 2:2 会生成 256*256 个子目录，keys_zone 是这个缓存空间的名字，cache 是用多少内存（这样热门的内容，Nginx 会直接放入内存，以提高访问速度），inactive 表示默认失效时间，max_size 表示最多用多少硬盘空间，需要注意的是 fastcgi_cache 缓存是先写在 fastcgi_temp_path 再移到 fastcgi_cache_path 中去的，所以这两个目录最好在同一个分区，从 0.8.9 之后可以放在不同的分区中，不过还是建议放在同一分区</p>
fastcgi_cache_valid	<p>示例：fastcgi_cache_valid 200 302 1h;</p> <p>用来指定应答代码的缓存时间，实例中的值表示将 200 和 302 应答缓存一个小时。</p> <p>示例：fastcgi_cache_valid 301 1d;</p> <p>将 301 应答缓存 1 天。</p> <p>示例：fastcgi_cache_valid any 1m;</p> <p>将其他应答缓存设置为 1 分钟</p>
fastcgi_cache_min_uses	<p>示例：fastcgi_cache_min_uses 1;</p> <p>设置请求几次之后响应将被缓存，1 表示一次即被缓存</p>
fastcgi_cache_use_stale	<p>示例：fastcgi_cache_use_stale error timeout invalid_header http_500;</p> <p>定义在哪些情况下使用过期缓存</p>
fastcgi_cache_key	<p>示例：fastcgi_cache_key \$request_method:\$host\$request_uri;fastcgi_cache_key http://\$host\$request_uri;</p> <p>定义 fastcgi_cache 的 key，示例中以请求的 URI 作为缓存的 key，Nginx 会取这个 key 的 md5 作为缓存文件，如果设置了缓存散列目录，Nginx 会从前往后取相应的位数作为目录。</p> <p>注意一定要加上 \$request_method 作为 cache key，否则如果先请求的为 head 类型，那么后面的 GET 请求将返回为空</p>

FastCGI Cache 资料见 [http://nginx.org/en/docs/http/nginx\\_http\\_fastcgi\\_module.html#fastcgi\\_cache](http://nginx.org/en/docs/http/nginx_http_fastcgi_module.html#fastcgi_cache)。

FastCGI 常见参数的 Nginx 配置示例如下：

```
[root@nginx conf]# cat nginx.conf
worker_processes 4;
worker_cpu_affinity 0001 0010 0100 1000;
worker_rlimit_nofile 65535;
user nginx;
events {
    use epoll;
    worker_connections 10240;
}
http {
    include mime.types;
    default_type application/octet-stream;
    sendfile on;
    tcp_nopush on;
    keepalive_timeout 65;
    tcp_nodelay on;
    client_header_timeout 15;
    client_body_timeout 15;
    send_timeout 15;

    log_format main '$remote_addr - $remote_user [$time_local] "$request" '
        '$status $body_bytes_sent "$http_referer" '
        '"$http_user_agent" "$http_x_forwarded_for"';

    server_tokens off;

    fastcgi_connect_timeout 240;
    fastcgi_send_timeout 240;
    fastcgi_read_timeout 240;
    fastcgi_buffer_size 64k;
    fastcgi_buffers 4 64k;
    fastcgi_busy_buffers_size 128k;
    fastcgi_temp_file_write_size 128k;
    #fastcgi_temp_path /data/nginx_fcgi_tmp;
    fastcgi_cache_path /data/nginx_fcgi_cache levels=2:2 keys_zone=nginx_fcgi_
cache:512m inactive=1d max_size=40g;
    #web.....
    server {
        listen 80;
        server_name blog.etiantian.org;
        root html/blog;
        location / {
            root html/blog;
            index index.php index.html index.htm;
        }
        location ~ /\. (php|php5)?$
        {
            fastcgi_pass 127.0.0.1:9000;
            fastcgi_index index.php;
```



```

include fastcgi.conf;
fastcgi_cache ngx_fcgi_cache;
fastcgi_cache_valid 200 302 1h;
fastcgi_cache_valid 301 1d;
fastcgi_cache_valid any 1m;
fastcgi_cache_min_uses 1;
fastcgi_cache_use_stale error timeout invalid_header http_500;
fastcgi_cache_key http://$host$request_uri;
}
access_log logs/web_blog_access.log main;
}
upstream blog_etiantian{
    server 10.0.0.8:8000 weight=1;
}
server {
    listen      8000;
    server_name blog.etiantian.org;
    location / {
        proxy_pass http://blog_etiantian;
        proxy_set_header Host $host;
        proxy_set_header X-Forwarded-For $remote_addr;
    }
    access_log logs/proxy_blog_access.log main;
}
}

```

为了让读者更加容易理解上述参数的作用及原理，我为大家画了一个原理图，如图 12-6 所示。

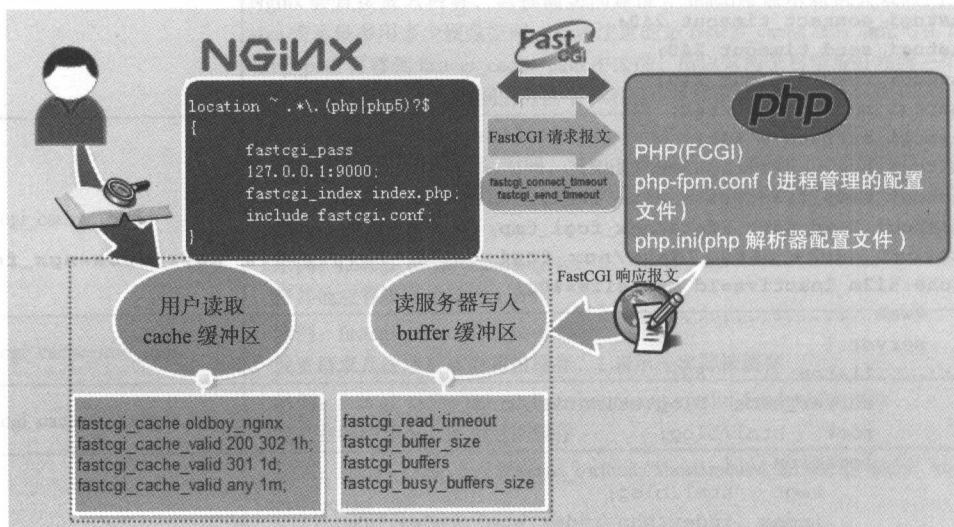


图 12-6 根据 HTTP 原理及 FastCGI 原理讲解 FastCGI 参数优化

Nginx 的 FastCGI 的相关参数和反向代理 proxy 的相关参数非常接近，大家在理解后文 Nginx 反向代理的参数时，可以结合此处 FastCGI 的参数进行对比理解。



## 12.2.11 配置 Nginx gzip 压缩实现性能优化

### 1. Nginx gzip 压缩功能介绍

Nginx gzip 压缩模块提供了压缩文件内容的功能，用户请求的内容在发送到用户客户端之前，Nginx 服务器会根据一些具体的策略实施压缩，以节约网站出口带宽，同时加快数据传输效率，来提升用户访问体验。

### 2. Nginx gzip 压缩的优点

- 提升网站的用户体验：发送给用户的内容小了，用户访问单位大小的页面就加快了，用户体验提升了，网站口碑就好了。
- 节约网站带宽成本：数据是压缩传输的，因此节省了网站的带宽流量成本，不过压缩时会稍微消耗一些 CPU 资源，这个一般可以忽略不计。

此功能既能提升用户体验，又能使公司少花钱，一举多得。对于几乎所有的 Web 服务来说，这是一个非常重要的功能，Apache 服务也有此功能。

### 3. 需要和不需要压缩的对象

- 纯文本内容的压缩比很高，因此，纯文本的内容最好进行压缩，例如：html、js、css、xml、shtml 等格式的文件。
- 被压缩的纯文本文件必须要大于 1KB，由于压缩算法的特殊原因，极小的文件压缩后可能反而变大。
- 图片、视频（流媒体）等文件尽量不要压缩，因为这些文件大多都是经过压缩的，如果再压缩很可能不会减小或减小得很少，甚至还有可能增大，同时压缩时还会消耗大量的 CPU、内存资源。

### 4. 参数介绍及配置说明

此压缩功能与早期 Apache 服务的 mod\_deflate 压缩功能很相似，Nginx 的 gzip 压缩功能依赖于 ngx\_http\_gzip\_module 模块，默认已安装。

对应的压缩参数说明如下：

```
# 压缩配置
gzip on;
#<== 开启 gzip 压缩功能。
gzip_min_length 1k;
#<== 设置允许压缩的最小页面字节数，页面字节数从 header 头的 Content-Length 中获取。默认值是 0，表示不管页面多大都进行压缩。建议设置成大于 1KB，如果小于 1KB 可能会越压越大。
gzip_buffers 4 16k;
#<== 压缩缓冲区大小。表示申请 4 个单位为 16KB 的内存作为压缩结果流缓存，默认值是申请与原始数据大小相同的内存空间来存储 gzip 压缩结果。
gzip_http_version 1.1;
#<== 压缩版本（默认为 1.1，前端为 squid2.5 时使用 1.0），用于设置识别 HTTP 协议版本，默认是 1.1，目前大部分浏览器已经支持 gzip 解压，使用默认值即可。
gzip_comp_level 2;
#<== 压缩比率。用来指定 gzip 压缩比，1 表示压缩比最小，处理速度最快；9 表示压缩比最大，传输速度
```

快，但处理最慢，也比较消耗 CPU 资源。

gzip\_types text/plain application/x-javascript text/css application/xml;  
#<== 用来指定压缩的类型，“text/html”类型总是会被压缩，这个就是 HTTP 原理部分所讲的媒体类型。

gzip\_vary on;  
#<==vary header 支持。该选项可以让前端的缓存服务器缓存经过 gzip 压缩的页面，例如用 Squid 缓存经过 Nginx 压缩的数据

完整的配置如下：

```
gzip on;  
gzip_min_length 1k;  
gzip_buffers 4 32k;  
gzip_http_version 1.1;  
gzip_comp_level 9;  
#gzip_types text/plain application/x-javascript text/css application/xml; #<==  
老的不正确写法  
gzip_types text/css text/xml application/javascript;  
gzip_vary on;
```

不同的 Nginx 版本中，gzip\_types 的配置可能会有所不同，上述配置示例适合 Nginx 1.6.3。对应的文件类型，请查看安装目录下的 mime.types 文件。

更多官方资料请看 [http://nginx.org/en/docs/http/nginx\\_http\\_gzip\\_module.html](http://nginx.org/en/docs/http/nginx_http_gzip_module.html)。

5. Nginx 压缩配置效果检查

可通过火狐浏览器加 Yslow 插件查看 gzip 压缩及 expires 缓存结果。提前安装好火狐浏览器，并且安装好 Yslow 插件，开启监控，然后打开 LNMP 时安装的博客地址，就可以看到如图 12-7 所示的压缩结果。

TYPE	SIZE (KB)	GZIP (KB)	COOKIE RECEIVED (bytes)	COOKIE SENT (bytes)	HEADERS	URL	EXPIRES (Y/M/D)
doc(1)	0.06K						
js(1)	72.1K						
js	72.1K	24.0K			ρ	http://blog.cntianian.org/oldboy/jquery.js	2012/10/20

图 12-7 检查 Nginx gzip 压缩效果图

6. 重要的前端网站调试工具介绍

常见的前端网站调试工具有如下几种。

- ❑ Google 浏览器（Chrome）：通过该浏览器直接按 F12 键即可查看压缩及缓存结果，另外，谷歌浏览器上也可以直接安装 Yslow 插件（如图 12-8 所示）。

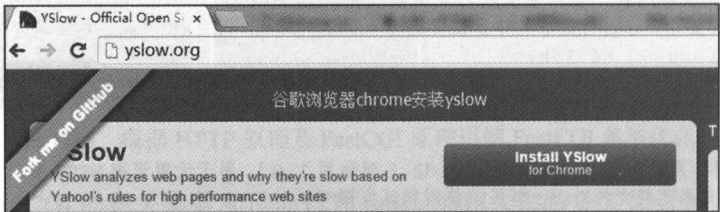


图 12-8 调试工具 yslow.org 网站

□ 火狐浏览器: 在该浏览器上安装 firebug、Yslow (<http://yslow.org/>), 即可进行调试 (如图 12-9 所示)。

□ IE 浏览器: 在该浏览器上安装 httpwatch 即可进行调试 (省略)。

## 12.2.12 配置 Nginx expires 缓存实现性能优化

### 1. Nginx expires 功能介绍

简单地说, Nginx expires 的功能就是为用户访问的网站内容设定一个过期时间, 当用户第一次访问这些内容时, 会把这些内容存储在用户浏览器本地, 这样用户第二次及以后继续访问该网站时, 浏览器会检查加载已经缓存在用户浏览器本地的内容, 就不会再去服务器下载了, 直到缓存的内容过期或被清除为止。

更深入地理解: expires 的功能就是允许通过 Nginx 配置文件控制 HTTP 的 "Expires" 和 "Cache-Control" 响应头部内容, 告诉客户端浏览器是否缓存和缓存多长时间以内访问的内容。这个 expires 模块控制 Nginx 服务器应答时的 expires 头内容和 Cache-Control 头的 max-age 指令。缓存的有效期可以设置为相对于源文件的最后修改时刻或客户端的访问时刻。

这些 HTTP 头向客户端表明了内容的有效性和持久性。如果客户端本地有内容缓存, 则内容就可以从缓存 (除非已经过期) 而不是从服务器中读取, 然后客户端会检查缓存中的副本, 看其是否过期或失效, 以决定是否重新从服务器中获得内容更新。

### 2. Nginx expires 作用介绍

在网站的开发和运营中, 视频、图片、CSS、JS 等网站元素的更改机会较少, 特别是图片, 这时可以将图片设置在客户浏览器本地缓存 365 天或 3650 天, 而将 CSS、JS、html 等代码缓存 10 ~ 30 天。这样用户第一次打开页面后, 会在本地的浏览器中按照过期日期缓存相应的内容, 下次用户再打开类似的页面时, 重复的元素就无需下载了, 从而加快了用户访问速度。用户的访问请求和数据减少了, 也可以节省大量的服务器端带宽。此功能同 Apache 的 expires 功能类似。

### 3. Nginx expires 功能优点

- expires 可以降低网站的带宽, 节约成本。
- 加快用户访问网站的速度, 提升用户的访问体验。
- 服务器访问量降低了, 服务器压力就减轻了, 服务器的成本也会降低, 甚至还可以节约人力成本。

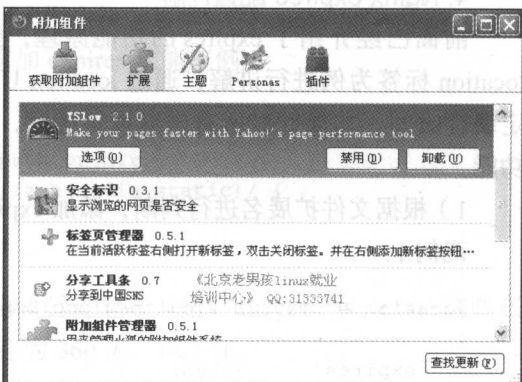


图 12-9 调试工具 Yslow 安装方法图

对于几乎所有的 Web 服务来说，这是非常重要的功能之一，Apache 服务也有此功能。

#### 4. Nginx expires 配置详解

前面已经介绍了 expires 的功能原理，接下来就来配置 Nginx expires 的功能。这里以 location 标签为例进行讲解，通过 location URI 规则将需要缓存的扩展名罗列出来，然后指定缓存时间。如果针对所有内容设置缓存，也可以不用 location。Nginx 默认安装了 expires 功能。

##### 1) 根据文件扩展名进行判断，添加 expires 功能范例

范例 1:

```
location ~ .*\. (gif|jpg|jpeg|png|bmp|swf)$
{
    expires      3650d;
}
```

该范例的意思是当用户访问网站 URL 结尾的文件扩展名为上述指定类型的图片时，设置缓存为 3650 天，即 10 年。

范例 2:

```
location ~ .*\. (js|css)?$
{
    expires      30d;
}
```

该范例的意思是当用户访问网站 URL 结尾的文件扩展名为 js、css 类型的元素时，设置缓存为 30 天，即 1 个月。

上述两个范例的实际配置结果如下：

```
###www
server {
    listen      80;
    server_name www.etiantian.org;
    location / {
        root    html/www;
        index   index.html index.htm;
    }
    location ~ .*\. (gif|jpg|jpeg|png|bmp|swf)$
    {
        expires      10y;
        root    html/www;
    }
    location ~ .*\. (js|css)?$
    {
        expires      30d;
    }
}
```

```

    access_log /app/logs/www_access.log main;
}

```

需要特别注意的是，location 内容一般要放到虚拟主机配置中，即 server 标签中。

2) 根据 URI 中的路径（目录）进行判断，添加 expires 功能范例

范例 3:

```

## Add expires header according to URI(path or dir).
location ~ ^/(images|javascript|js|css|flash|media|static)/ {
    expires 360d;
}

```

该范例的意思是当用户访问网站 URL 中包含上述的路径（例：images、js、css，这些在服务器端是程序目录）时，将访问的内容设置缓存为 360 天，即 1 年。

3) 单个文件添加 expires 功能的范例

下面的命令会给 robots.txt 机器人文件设置过期时间（这里为 7 天），在这 7 天内并不记录 404 错误日志。

```

location ~(robots.txt) {
    expires 7d;
    break;
}

```

## 5. Nginx expires 配置效果检查

检查 Nginx expires 的方法和检查 Nginx gzip 的方法相同。

通过火狐浏览器加 Yslow 插件查看 gzip 压缩及 expires 缓存结果时，要提前安装好火狐浏览器，并且要安装好 Yslow 插件，开启监控，然后打开 LNMP 时安装的博客地址（带有图片、JS、CSS），就可以看到如图 12-10 所示的缓存结果了。

I TYPE	SIZE (KB)	GZIP (KB)	COOKIE RECEIVED (bytes)	COOKIE SENT (bytes)	HEADERS	URL	EXPIRES (Y/M/D)	RESPONSE TIME (ms)	ETAG	ACTION
image (1)	35.1K									
image	35.1K		火狐浏览器安装Firebug, yslow插件			http://www.ctiantian.org/test1.jpg	2023/10/2	51		smush.it
favicon (1)	0.1K									

I TYPE	SIZE (KB)	GZIP (KB)	COOKIE RECEIVED (bytes)	COOKIE SENT (bytes)	HEADERS	URL	EXPIRES (Y/M/D)
doc (1)	0.06K						
js (1)	72.1K						
js	72.1K	24.0K				http://blog.ctiantian.org/oldboy/jquery.js	2012/10/20

图 12-10 查看 expires 是否生效图

在 Linux 客户端可通过如下 curl 命令查看图片 URL 缓存的 header 信息：

```
[root@oldboy ~]# curl -I http://blog.etiantian.org/23.jpg
HTTP/1.1 200 OK
Server: nginx
Date: Fri, 21 Sep 2012 02:07:18 GMT
Content-Type: image/jpeg
Content-Length: 59491
Last-Modified: Mon, 13 Aug 2012 02:26:18 GMT
Connection: keep-alive
ETag: "502865ca-e863"
Expires: Mon, 19 Sep 2022 02:07:18 GMT #<== 缓存的过期时间
Cache-Control: max-age=315360000 #<== 缓存的总时间，单位为秒
Accept-Ranges: bytes
```

## 6. Nginx expires 功能的缺点及解决方法

几乎所有的事物都有两面性，没有十全十美的人和事。Nginx expires 功能也不例外，虽然这个功能很好，但是也会给企业带来一些困惑。

当网站被缓存的页面或数据更新了，此时用户端看到的可能还是旧的已经缓存的内容，这样就会影响用户的体验，那么如何解决这个问题呢？解决办法如下。

第一，对于经常发生变动的图片等文件，可以缩短对象缓存时间，例如：谷歌和百度首页的图片经常会根据不同的日期换成一些节日的图片，所以这里可以将这个图片的缓存期设置为 1 天。

第二，当网站改版或更新时，可以在服务器中将缓存的对象改名（网站代码程序）。

- ❑ 对于网站的图片、附件，一般不会被用户直接修改，用户层面上的修改图片，实际上是重新传到服务器，虽然内容一样但是是一个新的图片名了。
- ❑ 网站改版升级会修改 JS、CSS 元素，若改版时对这些元素名进行了修改，则会使得前端的 CDN 及用户端需要重新缓存内容。

## 7. 企业网站缓存日期曾经的案例参考

若企业的业务和网站的访问量不同，那么网站缓存期的时间设置也是不同的，比如，如下企业所用的缓存日期就是不一样的。

- ❑ 51CTO：1 周
- ❑ 新浪：15 天
- ❑ 京东：25 年
- ❑ 淘宝：10 年

## 8. 企业网站有可能不希望被缓存的内容

- ❑ 广告图片，用于广告服务，都缓存了就不好控制展示了。
- ❑ 网站流量统计工具（JS 代码），都缓存了流量统计就不准了。
- ❑ 更新很频繁的文件（Google 的 logo），如果按天缓存，效果还是显著的。



## 12.3 Nginx 日志相关的优化与安全

### 12.3.1 编写脚本实现 Nginx access 日志轮询

当用户请求一个软件时，绝大多数软件都会记录用户的访问情况，Nginx 服务也不例外。Nginx 软件目前还没有类似 Apache 的通过 cronolog 或 rotatelog 对日志进行分割处理的功能，但是，运维人员可以利用脚本开发、Nginx 的信号控制功能或 reload 重新加载，来实现日志的自动切割、轮询。

详细操作过程如下。

(1) 配置日志切割脚本，命令如下：

```
[root@oldboy ~]# mkdir /server/scripts/ -p
[root@oldboy ~]# cd /server/scripts/
[root@oldboy scripts]# vim cut_nginx_log.sh
cd /application/nginx/logs &&\
/bin/mv www_access.log www_access_$(date +%F -d -1day).log
# <== 将日志按日期改成前一天的名称
/application/nginx/sbin/nginx -s reload # <== 重新加载nginx使得触发重新生成访问
日志文件
```

提示：实际上脚本的功能很简单，就是改日志名，然后加载 Nginx，重新生成文件记录日志

(2) 将这段脚本保存后加入到服务器端的定时任务配置里，使得此脚本在每天的凌晨 0 点执行，就可以实现日志每天的分割功能了，操作结果如下：

```
[root@oldboy scripts]# crontab -e #<== 打开编辑功能后，加入如下内容
#cut nginx access log by oldboy at 201409
00 00 * * * /bin/sh /server/scripts/cut_nginx_log.sh >/dev/null 2>&1
```

此处的意思是每天凌晨 0 点执行后面的 /server/scripts/cut\_nginx\_log.sh 脚本，>/dev/null 2>&1 表示任何输出都不要。这个是 Linux 的基础服务之一，具体请参考相关资料。

最终切割后的日志效果如下：

```
[root@oldboy scripts]# ll /application/nginx/logs/
总用量 27752
-rw-r--r--. 1 root root 20241716 10月 9 12:03 access.log
-rw-r--r--. 1 root root 27428 10月 10 04:22 error.log
-rw-r--r--. 1 root root 5 10月 9 09:56 nginx.pid
-rw-r--r--. 1 root root 264 9月 26 05:08 www_access_2014-09-25.log
-rw-r--r--. 1 root root 0 9月 26 05:12 www_access_2014-09-26.log
-rw-r--r--. 1 root root 0 9月 28 00:00 www_access_2014-09-27.log
-rw-r--r--. 1 root root 48717 9月 28 03:11 www_access_2014-09-28.log
-rw-r--r--. 1 root root 8072741 10月 9 17:25 www_access_2014-10-09.log
```



这里是按照不同的日期生成日志。

### 12.3.2 不记录不需要的访问日志

在实际工作中，对于负载均衡器健康节点的检查或某些特定文件（比如图片、JS、CSS）的日志，一般不需要记录下来，因为在统计 PV 时是按照页面进行计算的，而且日志如果写入太频繁会消耗大量的磁盘 I/O，降低服务的性能。

具体配置方法如下：

```
location ~ .*\. (js|jpg|JPG|jpeg|JPEG|css|bmp|gif|GIF)$ {
    access_log off;
}
```

这里用 location 标签匹配不记录日志的元素扩展名，然后关掉日志。

### 12.3.3 访问日志的权限设置

假如日志目录为 /app/logs，则授权方法如下：

```
chown -R root.root /app/logs
chmod -R 700 /app/logs
```

不需要在日志目录上给 Nginx 用户读或写的许可，但很多网友都没有注意这个问题，他们把该权限直接给了 Nginx 或 Apache 用户，这就造成了安全隐患。

## 12.4 Nginx 站点目录及文件 URL 访问控制

### 12.4.1 根据扩展名限制程序和文件访问

Web 2.0 时代，绝大多数网站都是以用户为中心的，例如：bbs、blog、sns 产品，这几个产品都有一个共同特点，那就是不但允许用户发布内容到服务器，还允许用户发图片甚至上传附件到服务器上，由于为用户开通了上传的功能，因此给服务器带来了很大的安全风险。虽然很多程序在上传前会对文件做一定的控制，例如：文件大小、类型等，但是，一不小心就会被黑客钻了空子，上传了木马程序。

下面将利用 Nginx 配置禁止访问上传资源目录下的 PHP、Shell、Perl、Python 程序文件，这样用户即使上传了木马文件也没法执行，从而加强了网站的安全。

范例 1：配置 Nginx，禁止解析指定目录下的指定程序。

```
location ~ ^/images/.*\. (php|php5|sh|pl|py)$
{
    deny all;
}
location ~ ^/static/.*\. (php|php5|sh|pl|py)$
{
    deny all;
}
```

```
location ~* ^/data/(attachment|avatar)/.*\.(php|php5)$
{
    deny all;
}
```

对上述目录的限制必须写在 Nginx 处理 PHP 服务配置的前面，如下：

```
location ~ .*\. (php|php5)?$
{
    fastcgi_pass 127.0.0.1:9000;
    fastcgi_index index.php;
    include fcgi.conf;
}
```

**范例 2：**Nginx 下配置禁止访问 \*.txt 和 \*.doc 文件。

实际配置信息如下：

```
location ~* \.(txt|doc)$ {
    if (-f $request_filename) {
        root /data/www/www;
        #rewrite .... 可以重定向到某个 URL
        break;
    }
}
location ~* \.(txt|doc){
    root /data/www/www;
    deny all;
}
```

## 12.4.2 禁止访问指定目录下的所有文件和目录

**范例 1：**配置禁止访问指定的单个或多个目录。

禁止访问单个目录的命令如下：

```
location ~ ^/(static)/ {
    deny all;
}
```

```
location ~ ^/static {
    deny all;
}
```

禁止访问多个目录的命令如下：

```
location ~ ^/(static|js) {
    deny all;
}
```

**范例 2：**禁止访问目录并返回指定的 HTTP 状态码，命令如下。

```
server {
```

```

listen      80;
server_name www.etiantian.org etiantian.org;
    root    /data0/www/www;
    index   index.html index.htm;
    access_log /app/logs/www_access.log commonlog;
    location /admin/ { return 404; }
    location /templates/ { return 403; }
}

```

作用：禁止访问目录下的指定文件，或者禁止访问指定目录下的所有内容。

最佳应用场景：对于集群的共享存储，一般是存放静态资源文件，所以可以禁止执行指定扩展名的程序，例如：.php、.sh、.pl、.py。

### 12.4.3 限制网站来源 IP 访问

下面介绍如何使用 ngx\_http\_access\_module 限制网站来源 IP 访问。

案例环境：phpmyadmin 数据库的 Web 客户端，内部开发人员所用的。

范例 1：禁止外界访问某目录，但允许某 IP 访问该目录，且支持 PHP 解析，命令如下。

```

location ~ ^/oldboy/ {
    allow 202.111.12.211;
    deny all;
}

location ~ .*\. (php|php5)?$ {
    fastcgi_pass 127.0.0.1:9000;
    fastcgi_index index.php;
    include fastcgi_params;
    fastcgi_param SCRIPT_FILENAME $document_root$fastcgi_script_name;
}
}

```

范例 2：限制指定 IP 或 IP 段访问，命令如下。

```

location / {
    deny 192.168.1.1;
    allow 192.168.1.0/24;
    allow 10.1.1.0/16;
    deny all;
}

```

参考：[http://nginx.org/en/docs/http/ngx\\_http\\_access\\_module.html](http://nginx.org/en/docs/http/ngx_http_access_module.html)。

企业问题案例：Nginx 做反向代理的时候可以限制客户端 IP 吗？

解答：可以，具体方法如下。

方法 1：使用 if 来控制，命令如下。

```

if ( $remote_addr = 10.0.0.7 ) {

```

```

return 403;
}
if ( $remote_addr = 218.247.17.130 ) {
    set $allow_access_root 'true';
}
http://nginx.org/en/docs/varindex.html

```

方法 2: 利用 deny 和 allow 只允许 IP 访问, 命令如下。

```

location / {
    root    html/blog;
    index  index.php index.html index.htm;
    allow  10.0.0.7;
    deny  all;
}

```

方法 3: 只拒绝某些 IP 访问, 命令如下。

```

location / {
    root    html/blog;
    index  index.php index.html index.htm;
    deny  10.0.0.7;
    allow  all;
}

```

#### 注意事项

- ❑ deny 一定要加一个 IP, 否则会直接跳转到 403, 不再往下执行了, 如果 403 默认页是在同一域名下, 那么会造成死循环访问。
- ❑ 对于 allow 的 IP 段, 将允许访问的段位从小到大排列, 如 127.0.0.0/24 的下面才能是 10.10.0.0/16, 其中:
  - 24 表示子网掩码: 255.255.255.0。
  - 16 表示子网掩码: 255.255.0.0。
  - 8 表示子网掩码: 255.0.0.0。
- ❑ 以 “deny all;” 结尾, 表示除了上面允许的, 其他的都禁止。如:

```

deny 192.168.1.1;
allow 127.0.0.0/24;
allow 192.168.0.0/16;
allow 10.10.0.0/16;
deny all;

```

### 12.4.4 配置 Nginx, 禁止非法域名解析访问企业网站

问题: Nginx 如何防止用户 IP 访问网站 (恶意域名解析, 也相当于是直接 IP 访问企业网站)?

方法 1: 让使用 IP 访问网站的用户, 或者恶意解析域名的用户, 收到 501 错误, 命令

如下。

```
server {  
    listen 80 default_server;  
    server_name _;  
    return 501;  
}
```



说明 直接报 501 错误，从用户体验上不是很好。

方法 2：通过 301 跳转到主页，命令如下。

```
server {  
    listen 80 default_server;  
    server_name _;  
    rewrite ^(.*) http://blog.etiantian.org/$1 permanent;  
}
```

方法 3：发现某域名恶意解析到公司的服务器 IP，在 server 标签里添加以下代码即可，若有多个 server 则要多处添加。

```
if ($host !~ ^www/.eduoldboy/.com$){  
    rewrite ^(.*) http://www.eduoldboy.com$1 permanent;  
}
```

上面代码的意思是如果 header 信息的 host 主机名字段非 www.eduoldboy.com，就 301 跳转到 www.eduoldboy.com。

## 12.5 Nginx 图片及目录防盗链解决方案

### 1. 什么是资源盗链

简单地说，就是某些不法网站未经许可，通过在其自身网站程序里非法调用其他网站的资源，然后在自己的网站上显示这些调用的资源，达到填充自身网站的效果。这一举动不仅浪费了调用资源网站的网络流量，还造成其他网站的带宽及服务压力吃紧，甚至宕机。

下面通过示意图阐述资源盗链的原理，如图 12-11 所示。

### 2. 网站资源被盗链带来的问题

若网站图片及相关资源被盗链，最直接的影响就是网络带宽占用加大了，带宽费用多了，网络流量也可能忽高忽低，Nagios/Zabbix 等报警服务频繁报警，类似图 12-12 所示。

最严重的情况就是网站的资源被非法使用，使网站带宽成本加大和服务器压力加大，这有可能会造成数万元的损失，且网站的正常用户访问也会受到影响。



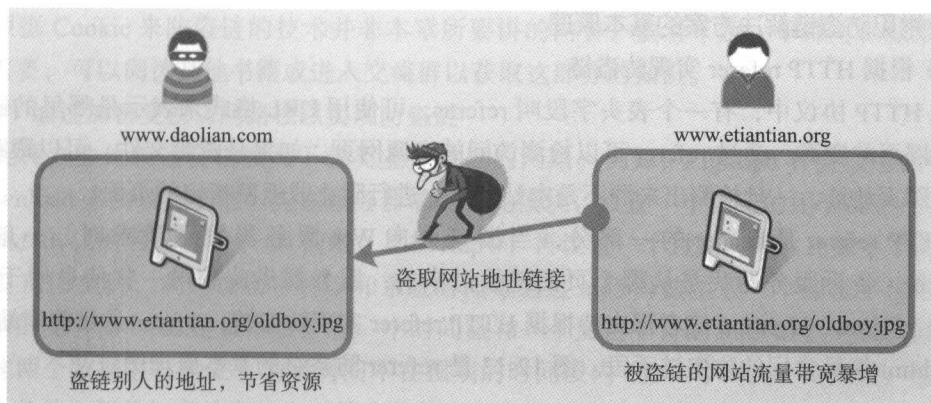


图 12-11 资源盗链原理的详细示意图

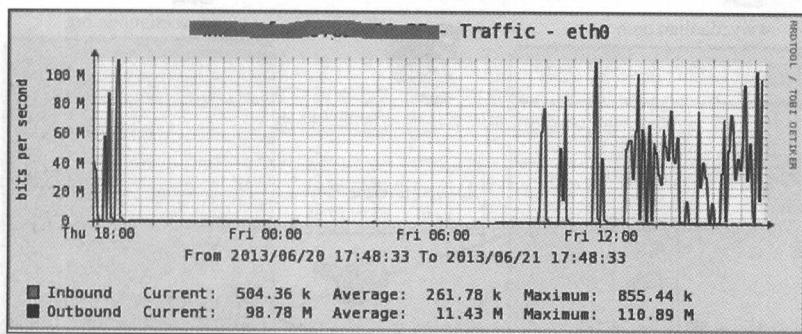


图 12-12 因盗链导致的流量飙升图

### 3. 企业真实案例：网站资源被盗链，出现严重问题

某日，我接到从事运维工作的朋友的紧急求助，其公司的 CDN 源站的流量没有变动，但 CDN 加速那边的流量无故超出了好几个 GB，不知道该怎么处理。

该故障的影响：由于是购买的 CDN 网站加速服务，因此虽然流量多了几个 GB，但是业务未受影响。只是，这么大的异常流量，持续下去可直接导致公司无故损失数万元。解决这个问题可体现运维的价值。

那么该如何及时发现，又如何处理这样的问题呢？

本节先给大家讲述几个发现问题的方法，如何处理盗链，后文会详细讲解。

第一，对 IDC 及 CDN 带宽做监控报警。

第二，作为高级运维或运维经理，每天上班的重要任务，就是经常查看网站流量图，关注流量变化，关注异常流量。

第三，对访问日志做分析，迅速定位异常流量，并且和公司市场推广等保持较好的沟通，以便调度带宽和服务器资源，确保网站正常的访问体验。

更多企业案例及实战解决方案请参见：<http://oldboy.blog.51cto.com/2561410/909696>。

#### 4. 常见防盗链解决方案的基本原理

##### 1) 根据 HTTP referer 实现防盗链

在 HTTP 协议中，有一个表头字段叫 referer，可使用 URL 格式来表示是哪里的链接用了当前网页的资源。通过 referer 可以检测访问的来源网页，如果是资源文件，可以跟踪到显示它的网页地址，一旦检测出来源不是本站，马上进行阻止或返回指定的页面。

HTTP referer 是 header 的一部分，当浏览器向 Web 服务器发送请求时，一般会带上 referer，告诉服务器我是从哪个页面链接过来的，服务器借此获得一些信息用于处理。Apache、Nginx、Lighttpd 三者都支持根据 HTTP referer 实现防盗链，referer 是目前网站图片、附件、html 等最常用的防盗链手段。图 12-13 是 referer 防盗链的基本原理图。

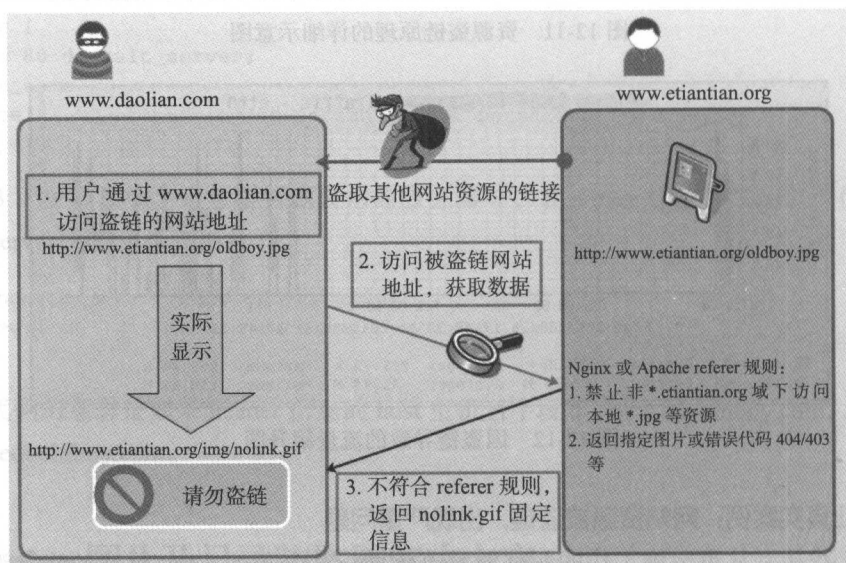


图 12-13 通过 referer 解决盗链问题的基本原理图

##### 2) 根据 Cookie 防盗链

对于一些特殊的业务数据，例如流媒体应用通过 ActiveX 显示的内容（例如，Flash、Windows Media 视频、流媒体的 RTSP 协议等），因为它们不向服务器提供 referer header，所以若采用上述的 referer 防盗链手段，将会达不到想要的效果。

对于 Flash、Windows Media 视频这种占用流量较大的业务数据，防盗链是比较困难的，此时可以采用 Cookie 技术，解决 Flash、Windows Media 视频等的防盗链问题。

例如：ActiveX 插件不传递 referer，但会传递 Cookie，可以在显示 ActiveX 的页面的 `<head>` `</head>` 标签内嵌入一段 JavaScript 代码，设置 “Cookie: Cache=av”，代码如下：

```
<script> document.cookie="Cache=av;domain=domain.com;path=/"; </script>
```

然后就可以通过各种手段来判断这个 Cookie 是否存在，以及验证其值的操作了。

根据 Cookie 来防盗链的技术并非本章所要讲的内容,读者了解一下即可,如果企业确实有需要,可以阅读其他书籍或进入交流群以获取这部分的知识。

### 3) 通过加密变换访问路径以实现防盗链

此种方法比较适合视频及下载类业务数据的网站。例如:Lighttpd 有类似的插件 `mod_secdownload` 来实现此功能。先在服务器端配置此模块,设置一个固定的用于加密的字符串,比如 `oldboy`,然后设置一个 url 前缀,比如 `/mp4/`,再设置一个过期时间,比如 1 小时,然后写一段 PHP 代码,利用加密字符串和系统时间等通过 md5 算法生成一个加密字符串。最终获取到的文件的 URL 链接中会带有一个时间戳和一个加密字符的 md5 数值,在访问时系统会对这两个数据进行验证。如果时间不在预期的时间段内(如 1 小时内)则失效;如果时间戳符合条件,但是加密的字符串不符合条件也会失效,从而达到防盗链的效果。

PHP 代码实例如下:

```
<?php
$secret = "oldboy";           // 加密字符串,必须和 lighttpd.conf 里的保持一致
$uri_prefix = "/mp4/";       // 虚拟的路径、前缀,必须和 lighttpd.conf 里的保持一致
$file = "/test.mp4";         // 实际文件名,必须加 "/" (斜杠)
$timestamp = time();          // current timestamp
$st_hex = sprintf("%08x", $timestamp);
$m = md5($secret.$file.$st_hex);
printf('%s', $uri_prefix, $m, $st_hex, $file, $file); // 生成 url 地址串
?>
```

根据 Lighttpd 的插件 `mod_secdownload` 来进行防盗链的技术并非本章所要讲的内容,读者了解一下即可,如果企业确实有需要,可以阅读其他书籍,或者进入交流群以获取这部分的知识。

## 5. Nginx Web 服务实现防盗链实战

在默认情况下,只需要进行简单的配置,即可实现防盗链处理。请看下面的实例。

### 1) 利用 referer, 并且针对扩展名 rewrite 重定向

下面的代码为利用 `referer` 且针对扩展名 `rewrite` 重定向,即实现防盗链的 Nginx 配置:

```
location ~* \.(jpg|gif|png|swf|flv|wma|wmv|asf|mp3|mmf|zip|rar)$ {
    valid_referers none blocked *.etiantian.org etiantian.org;
    if ($invalid_referer) {
        rewrite ^/ http://www.etiantian.org/img/nolink.jpg;
    }
}
```

 **提示** 要根据自己的实际业务(是否有外链的合作),进行域名设置。

设置 `expires` 的方法如下:

```
[root@oldboy bbs]# cat /application/nginx/conf/extra/www.conf
server {
    listen      80;
    server_name www.etiantian.org;
    root        html/www;
    index        index.html index.htm;
    access_log   logs/www_access.log main;
#Preventing hot linking of images and other file types
location ~* ^.+\. (gif|jpg|png|swf|flv|rar|zip)$ {
    valid_referers none blocked server_names *.etiantian.org etiantian.org;
    if ($invalid_referer) {
        rewrite ^/ http://bbs.etiantian.com/img/nolink.jpg;
    }
    access_log off;
    root html/www;
    expires 1d;
    break;
}
}
```

## 2) 利用 referer，并且针对站点目录过滤返回错误码

针对目录的方法如下：

```
location /images {
    root /data0/www/www;
    valid_referers none blocked *.etiantian.org etiantian.org;
    if ($invalid_referer){
        return 403;
    }
}
```

在上面这段防盗链设置中，分别针对不同的文件类型和不同的目录进行了设置，读者可以根据自己的需求进行类似设定。下面是上述代码的说明：

- ❑ “jpg|gif|png|swf|flv|wma|wmv|asf|mp3|mmf|zip|rar” 表示对以 .jpg、.gif、.png、.swf、.flv、.wma、.wmv、.asf、.mp3、.mmf、.zip 和 .rar 为后缀的文件实行防盗链处理。
- ❑ “\*.etiantian.org etiantian.org” 表示这个请求可以正常访问上面指定的文件资源。
- ❑ if{} 中内容的意思是如果地址不是上面指定的地址就跳转到通过 rewrite 指定的地址，也可以直接通过 return 返回 403 错误。
- ❑ return 403 为自定义的 http 返回状态码。
- ❑ “rewrite ^/ http://www.etiantian.org/img/nolink.jpg;” 表示显示一张防盗链图片。
- ❑ “access\_log off;” 表示不记录访问日志，减轻压力。
- ❑ expires 3d 指的是对所有文件设置 3 天的浏览器缓存。

## 6. NginxHttpAccessKeyModule 实现防盗链介绍

如果不怕麻烦，有条件实现的话，推荐使用 NginxHttpAccessKeyModule。

其运行方式是：如果 download 目录下有一个 file.zip 的文件。对应的 URI 是 <http://www.abc.com/download/file.zip>，使用 ngx\_http\_accesskey\_module 模块后就成了 <http://www.abc.com/download/file.zip?key=09093abeac094>，只有正确地给定了 key 值，才能下载 download 目录下的 file.zip，而且 key 值是与用户的 IP 相关的，这样就可以避免被盗链了。据说，现在 NginxHttpAccessKeyModule 连迅雷都可以防了，读者可以尝试一下。

### 7. 在产品设计上解决盗链方案

产品在设计时，处理盗链问题可以将计就计，为网站上传的图片增加水印。例如，图 12-14 就是为网站上传的图片增加的水印。



为图片添加版权水印是很有效的方法。网站直接转载图片一般是为了快捷，但是对于有水印的图片，很多站长是不愿意转载的。

图 12-14 网站被盗链后的提示  
图片示例

### 8. Nginx 防盗链花絮

下面实战模拟演示盗链。

(1) 假定 blog.etiantian.com 是非法盗链的网站域名，先编写如下的 oldboy.html 程序：

```
<html>
<head>
<title>
老男孩教育
</title>
</head>
<body bgcolor=green>
老男孩的博客！<br>我的博客是<a href="http://oldboy.blog.51cto.com" target="_blank">
博客地址</a>

</body>
</html>
```

这个非法盗链的访问地址是 <http://blog.etiantian.com/oldboy.html>，网站里会加载 [www.etiantian.org](http://www.etiantian.org) 网站的图片 stu.jpg，即盗用该网站图片并消耗了 [www.etiantian.org](http://www.etiantian.org) 正常网站的带宽。

(2) 假定我们维护的网站为 [www.etiantian.org](http://www.etiantian.org)，设定一张存在的图片的地址为 [www.etiantian.org/stu.jpg](http://www.etiantian.org/stu.jpg)，此时发现被 [blog.etiantian.com](http://blog.etiantian.com) 盗链了。通过查看流量，分析日志查看 referer，就可以看到被盗链的具体情况。

下面是查看 Web 用户的 http\_referer 的实战演示。

Nginx 日志格式为 [www.etiantian.org](http://www.etiantian.org)，其内容如下：

```
log_format main '$remote_addr - $remote_user [$time_local] "$request" '
'$status $body_bytes_sent "$http_referer" ';
```

```
"$http_user_agent" "$http_x_forwarded_for";
```

当盗链的网站 [blog.etiantian.com](http://blog.etiantian.com) 访问我们的站点时，记录的日志如下：

```
[root@nginx conf]# tail -f ../logs/www_access.log
10.0.0.125 - - [10/Mar/2015:19:34:15 +0800] "GET /stu.jpg HTTP/1.1" 200 68080
"http://blog.etiantian.com/oldboy.html" "Mozilla/5.0 (Windows NT 6.1; WOW64;
rv:29.0) Gecko/20100101 Firefox/29.0" "-"
```

我们自己正常访问用户网站时，站点记录的日志如下：

```
10.0.0.125 - - [10/Mar/2015:19:37:45 +0800] "GET /img/nolink.jpg HTTP/1.1" 304
0 "-" "Mozilla/4.0 (compatible; MSIE 8.0; Windows NT 6.1; WOW64; Trident/4.0;
SLCC2; .NET CLR 2.0.50727; .NET CLR 3.5.30729; .NET CLR 3.0.30729; Media Center
PC 6.0; .NET4.0C; .NET4.0E; InfoPath.3)" "-"
```

对比可知，盗链的网站多出了一个 `http_referer` 的信息，即“<http://blog.etiantian.com/oldboy.html>”，表示使用我们网站资源的用户来自于该网站，也就是该站点盗链了网站 [www.etiantian.org](http://www.etiantian.org) 资源。

可在 [www.etiantian.org](http://www.etiantian.org) 网站下设置防盗链，Nginx 的方法如下：

```
#Preventing hot linking of images and other file types
location ~* ^.+\. (jpg|png|swf|flv|rar|zip)$ {
    valid_referers none blocked *.etiantian.org etiantian.org;
    if ($invalid_referer) {
        rewrite ^/ http://bbs.etiantian.org/img/nolink.gif;
    }
    root html/www;
}
```



如果 `referers` 不是“`*.etiantian.org etiantian.org;`”，给它一个 <http://bbs.etiantian.org/img/nolink.gif>。

此外，给盗链网站显示的图片域名不要和被盗链的网站域名一样。也就是 <http://bbs.etiantian.org/img/nolink.gif> 里面的 `bbs.etiantian.org` 不能是 `www.etiantian.org`。

(3) 单独搭建一个虚拟机主机 `bbs.etiantian.org`，指定盗链后展示的图片为 <http://bbs.etiantian.org/img/nolink.gif>。

上面演示的盗链操作步骤汇总如下：

- (1) 将 [www.etiantian.org/stu.jpg](http://www.etiantian.org/stu.jpg) 指定被盗链的图片地址。
- (2) 将 <http://bbs.etiantian.org/img/nolink.gif> 指定盗链后展示的图片。
- (3) 输入非法盗链网站提供给用户的 URL，为 <http://blog.etiantian.com/oldboy.html>。

在没有设置盗链之前，显示 [www.etiantian.org/stu.jpg](http://www.etiantian.org/stu.jpg)，设置盗链之后，显示 <http://bbs.etiantian.org/img/nolink.gif> 对应的图片，但地址还是盗链的地址。



## 12.6 Nginx 错误页面的优雅显示

### 12.6.1 生产环境中常见的 HTTP 状态码列表

企业生产环境中常见的 HTTP 状态码列表如表 12-2 所示。

表 12-2 常见的 HTTP 状态码列表

状态代码及英文描述	代码描述
200 - OK - Standard response for successful HTTP requests.	服务器成功返回网页，这是成功的 HTTP 请求返回的标准状态码
301 - Moved Permanently - This and all future requests should be directed to the given.	永久跳转，所有请求的网页将永久跳转到被设定的新位置，例如：从 etiantian.org 跳转到 www.etiantian.org
403 - Forbidden - forbidden request (matches a deny filter) => HTTP 403 - The request was a legal request, but the server is refusing to respond to it.	禁止访问，这个请求是合法的，但是服务器端因为匹配了预先设置的规则而拒绝响应客户端的请求，此类问题一般为服务器权限配置不当所致
404 - Not Found - The requested resource could not be found but may be available again in the future.	服务器找不到客户端请求的指定页面，可能是客户端请求了服务器不存在的资源所导致的
500 - Internal Server Error - internal error in haproxy => HTTP 500 - A generic error message, given when no more specific message is suitable.	内部服务器错误，服务器遇到了意料之外的情况，不能完成客户的请求。这是一个较为笼统的报错，一般为服务器的设置或内部程序问题所致
502 - Bad Gateway - the server returned an invalid or incomplete response => HTTP 502 - The server was acting as a gateway or proxy and received an invalid response from the upstream server.	坏的网关，一般是代理服务器请求后端服务时，后端服务不可用或没有完成响应网关服务器。一般为代理服务器下面的节点出了问题所致
503 - Service Unavailable - no server was available to handle the request => HTTP 503 - The server is currently unavailable (because it is overloaded or down for maintenance).	服务当前不可用，可能为服务器超载或停机维护所致，或者是代理服务器后面没有可以提供服务的节点
504 - Gateway Timeout - the server failed to reply in time => HTTP 504 - The server was acting as a gateway or proxy and did not receive a timely response from the upstream server.	网关超时，一般是网关代理服务器请求后端服务时，后端服务没有在特定的时间内完成处理请求，一般为服务器过载所致，没有在指定的时间内返回数据给代理服务器

参考资料请见 <http://oldboy.blog.51cto.com/2561410/716294>。

### 12.6.2 为什么要配置错误页面优雅显示

在网站的运行过程中，可能因为页面不存在或系统过载等原因，导致网站无法正常响应用户的请求，此时 Web 服务会返回系统默认的错误码，或者很不友好的页面（如图 12-15 所示）。



图 12-15 访问不到内容时出现的 404 界面

我们可以将 404、403 等的错误信息页面重定向到网站首页或其他事先指定的页面，以提升网站的用户访问体验。

例如：老男孩的博客所在的网站，在这方面的处理就不是很严谨，给了用户不好的页面体验。

范例 1：对错误代码 403 实行本地页面跳转，命令如下。

```
###www
server {
    listen      80;
    server_name www.etiantian.org;
    location / {
        root    html/www;
        index   index.html index.htm;
    }
    error_page 403 /403.html; #<== 当出现 403 错误时，会跳转到 403.html 页面
}
```

上面的 /403.html 是相对于站点根目录 html/www 的。

范例 2：对错误代码 404 实行本地页面优雅显示，命令如下。

```
server {
    listen      80;
    server_name www.oldboyedu.com;
    location / {
        root    html/www;
        index   index.html index.htm;
        error_page 404 /404.html;
        #<== 当出现 404 错误时，会跳转到 404.html 页面
        access_log /app/logs/bbs_access.log commonlog;
    }
}
```

代码中的 /404.html 是相对于站点根目录 html/www 的。

范例 3：50x 页面放到本地单独目录下，进行优雅显示。

```
# redirect server error pages to the static page /50x.html
error_page 500 502 503 504 /50x.html;
location = /50x.html {
    root /data0/www/html;
}
```

这里指定单独的站点目录存放到 50x.html 文件中。

范例 4：改变状态码为新的状态码，并显示指定的文件内容，命令如下。

```
error_page 404 =200 /empty.gif;
server {
    listen 80;
    server_name www.linuxpeixun.com;
    location / {
        root /data0/www/bbs;
        index index.html index.htm;
        fastcgi_intercept_errors on;
        error_page 404 =200 /ta.jpg;
        access_log /app/logs/bbs_access.log commonlog;
    }
}
```

范例 5：错误状态码 URL 重定向，命令如下。

```
server {
    listen 80;
    server_name www.oldboyedu.com;
    location / {
        root html/www;
        index index.html index.htm;
        error_page 404 http://oldboy.blog.51cto.com;
        #<== 当出现 404 错误时，会跳转到指定的 URL http://oldboy.blog.51cto.com 页面显示
        给用户，这个 URL 一般是企业另外的可用地址
        access_log /app/logs/bbs_access.log commonlog;
    }
}
```

代码中的 /404.html 是相对于站点根目录 html/www 的。

范例 6：将错误状态码重定向到一个 location，命令如下。

```
location / {
    error_page 404 = @fallback;
}
location @fallback {
    proxy_pass http://backend;
}
```

官方说明如下：

```
syntax: error_page code ... [=[response]] uri;
default: —
```

```
context: http, server, location, if in location
```

Defines the URI that will be shown for the specified errors. These directives are inherited from the previous level if and only if there are `noerror_pagedirectives` on the current level. A `uri` value can contain variables.

Example:

```
error_page 404 /404.html;
```

```
error_page 500 502 503 504 /50x.html;
```

Furthermore, it is possible to change the response code to another using the `"=code"` syntax, for example:

```
error_page 404 =200 /empty.gif;
```

If an error response is processed by a proxied server, or a FastCGI server, and the server may return different response codes (e.g., 200, 302, 401 or 404), it is possible to respond with a returned code:

```
error_page 404 = /404.php;
```

It is also possible to use redirects for error processing:

```
error_page 403 http://example.com/forbidden.html;
```

```
error_page 404 =301 http://example.com/notfound.html;
```

In this case, the response code 302 is returned to the client. It can only be changed to one of the redirect status codes (301, 302, 303 and 307).

If there is no need to change URI during internal redirection it is possible to pass error processing into a named location:

```
location / {
    error_page 404 = @fallback;
}
```

```
location @fallback {
    proxy_pass http://backend;
}
```

If the `uri` processing led to an error, the HTTP status code indicating last occurred problem will be returned.

阿里门户网站天猫的 Nginx 优雅显示配置案例如下：

```
error_page 500 501 502 503 504 http://err.tmall.com/error2.html;
error_page 400 403 404 405 408 410 411 412 413 414 415 http://err.tmall.com/
error1.html;
```

## 12.7 Nginx 站点目录文件及目录权限优化

### 1. 单机 LNMP 环境目录权限严格控制措施

为了保证网站不遭受木马入侵，所有站点目录的用户和组都应该为 `root`，所有的目录权限都是 `755`；所有的文件权限都是 `644`。设置如下：

```
[root@www ~]# ls -l /var/html/blog/|tail -5
-rw-r--r-- 1 root root 7712 5月 2 2012 wp-mail.php
-rw-r--r-- 1 root root 9916 4月 27 2012 wp-settings.php
-rw-r--r-- 1 root root 18299 4月 21 2012 wp-signup.php
-rw-r--r-- 1 root root 3700 1月 9 2012 wp-trackback.php
-rw-r--r-- 1 root root 2788 2月 17 2012 xmlrpc.php
```

以上的权限设置可以防止黑客上传木马, 以及修改站点文件, 但是, 合理的网站用户上传的内容也会被拒之门外。那么如何让合法的用户可以上传文件, 而又不至于被黑客利用攻击呢?

如果是单机的 LNMP 环境, 那么站点目录和文件属性的设置如下。

先把所有的目录权限设置为 755, 所有的文件权限设置为 644, 所用目录和文件的用户和组都是 root; 然后把用户上传资源的目录权限设置为 755, 将用户和组设置为 Nginx 服务的用户; 最后针对上传资源的目录做资源访问限制 (前文已述)。

部分公司所采用的授权方式不是很安全, 常见的有如下两种:

❑ `chmod -R 777 /sitedir`

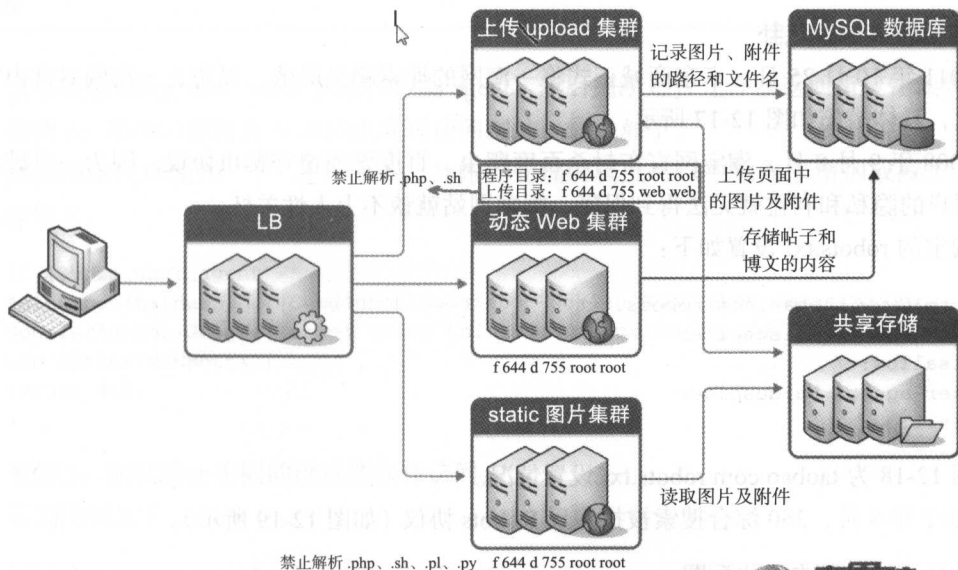
❑ `chown -R nginx.nginx /sitedir`

上述两种授权方法虽然不能说是错误的, 但是没有做到授权最小化, 会给网站带来非常大的安全隐患, 特别是木马入侵的时候。

在比较好的网站业务架构中, 应把资源文件, 包括用户上传的图片、附件等服务和程序服务分离, 最好把上传程序服务也分离出来, 这样就可以从容地按照上文所述的方法进行安全授权了。

## 2. Nginx 企业网站集群超级安全设置

结合 Linux 权限体系及 Nginx 大型集群架构进行配置, 严格控制针对 Nginx 目录的访问才能降低网站被入侵的风险。比如, 可根据图 12-16 中的企业集群架构逻辑图 and 不同角色提供的不同服务来严格控制不同服务器的 Nginx 目录权限。



老男孩教育 · 企业标准网站架构优化逻辑图



图 12-16 专业的 Nginx 集群架构角色逻辑图

表 12-3 为集群架构中不同于前面 Web 业务的权限管理细化。

表 12-3 集群架构中不同角色的授权具体思路说明

服务器角色	权限处理	安全系数
动态 Web 集群	目录权限 755，文件权限 644，所用的目录和文件的用户和组都是 root。环境为 Nginx+PHP	文件不能被更改，目录不能被写入，安全系数 10
static 图片集群	目录权限 755，文件权限 644，所用的目录和文件的用户和组都是 root。环境为 Nginx	文件不能被更改，目录不能被写入，安全系数 10
上传 upload 集群	目录权限 755，文件权限 644，所用的目录和文件的用户和组都是 root。特别注意的是：用户上传的目录设置为 755，用户和组均使用 Nginx 服务配置的用户	文件不能被更改，目录不能被写入，但是用户上传的目录允许写入文件且需要通过 Nginx 的其他功能来禁止读文件，安全系数 8

做到上述的设置后，网站服务在系统层面被入侵的风险就大大降低了。

12.8 Nginx 防爬虫优化

1. Robots.txt 机器人协议介绍

Robots 协议（也称为爬虫协议、机器人协议等）的全称是“网络爬虫排除标准”（Robots Exclusion Protocol），网站通过 Robots 协议告诉搜索引擎哪些页面可以抓取，哪些页面不能抓取。有关 robots.txt 的配置，本书不会详细讲解。

2. 机器人协议八卦

2011 年 10 月 25 日，京东商城正式将一淘网的搜索爬虫屏蔽，以防止一淘网对其内容进行抓取，具体措施如图 12-17 所示。

2008 年 9 月 8 日，淘宝网宣布封杀百度爬虫，百度忍痛遵守爬虫协议。因为一旦破坏协议，用户的隐私和利益就无法得到保障，搜索网站就谈不上人性关怀。

淘宝的 robots.txt 设置如下：

```
http://www.taobao.com/robots.txt
User-agent: Baiduspider
Disallow: /
User-agent: baiduspider
Disallow: /
```

图 12-18 为 taobao.com robots.txt 设置情况。

2012 年 8 月，360 综合搜索被指违反 Robots 协议（如图 12-19 所示）。

3. Nginx 防爬虫优化配置

我们可以根据客户端的 user-agents 信息，轻松地阻止指定的爬虫爬取我们的网站。下面来看几个案例。



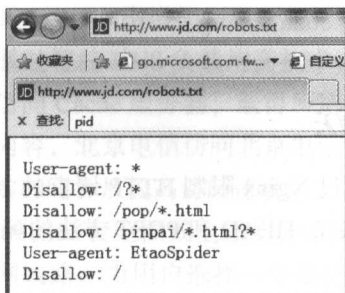


图 12-17 jd.com robots.txt 设置情况

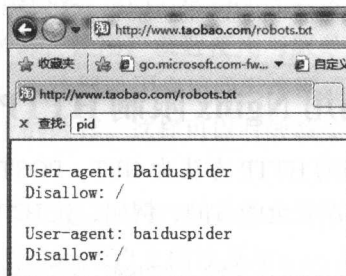


图 12-18 taobao.com robots.txt 设置情况

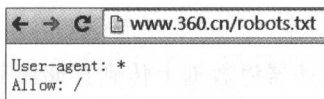


图 12-19 360.cn robots.txt 设置情况

范例 1：阻止下载协议代理，命令如下。

```
## Block download agents ##
if ($http_user_agent ~* LWP::Simple|BBBike|wget) {
    return 403;
}
```



说明 如果用户匹配了 if 后面的客户端（例如 wget），就返回 403。

这里根据 \$http\_user\_agent 获取客户端 agent，然后判断是否允许或返回指定错误码。

范例 2：添加内容防止 N 多爬虫代理访问网站，命令如下。

这些爬虫代理使用“|”进行分隔，具体要处理的爬虫可以根据需求增加或减少，添加的内容如下：

```
if ($http_user_agent ~*
"qihoobot|Baiduspider|Googlebot|Googlebot-Mobile|Googlebot-Image|Mediapartners-
Google|Adsbot-Google|Yahoo! Slurp China|YoudaoBot|Sosospider|Sogou spider|Sogou
web spider|MSNBot") {
    return 403;
}
```

范例 3：测试禁止不同的浏览器软件访问。

示例代码如下：

```
if ($http_user_agent ~* "Firefox|MSIE")
{
    rewrite ^(.*) http://blog.etiantian.org/$1 permanent;
}
```

如果浏览器为 Firefox 或 IE，就会跳转到 <http://blog.etiantian.org>。

## 12.9 利用 Nginx 限制 HTTP 的请求方法

最常用的 HTTP 方法为 GET、POST，我们可以通过 Nginx 限制 HTTP 请求的方法来达到提升服务器安全的目的，例如，让 HTTP 只能使用 GET、HEAD 和 POST 方法的配置如下：

```
#Only allow these request methods
if ($request_method !~ ^(GET|HEAD|POST)$ ) {
    return 501;
}
#Do not accept DELETE, SEARCH and other methods
```

12.7 节中提到过，当上传服务器将数据上传到存储服务器时，用户上传写入的目录就不得不给 Nginx 对应的用户赋予相关权限，这样一旦程序出现漏洞，木马就有可能被上传到服务器挂载的对应存储服务器的目录里，虽然我们也做了禁止 PHP、SH、PL、PY 等扩展名的解析限制，但还是会遗漏一些意想不到的可执行文件。对于这种情况，该怎么办呢？事实上，还可以通过限制上传服务器的 Web 服务（可以具体到文件）使用 GET 方法，防止用户通过上传服务器访问存储内容，让访问存储渠道只能从静态或图片服务器入口进入。例如，在上传服务器上限制 HTTP 的 GET 方法的配置如下：

```
## Only allow GET request methods ##
if ($request_method ~* ^(GET)$ ) {
    return 501;
}
```



提示 还可以加一层 location，更具体地限制文件名。

实际效果如图 12-20 所示。

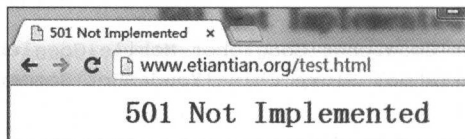


图 12-20 禁止 HTTP GET 方法的实际访问效果

## 12.10 使用 CDN 做网站内容加速

### 12.10.1 什么是 CDN

CDN 的全称是 Content Delivery Network，中文意思是内容分发网络。简单地讲，通过

在现有的 Internet 中增加一层新的网络架构，将网站的内容发布到最接近用户的 Cache 服务器内，通过智能 DNS 负载均衡技术，判断用户的来源，让用户就近使用与服务器相同线路的带宽访问 Cache 服务器，取得所需的内容。例如：天津网通用户访问天津网通 Cache 服务器上的内容，北京电信访问北京电信 Cache 服务器上的内容。这样可以有效减少数据在网络上传输的时间，从而提高访问速度。

CDN 是一套全国或全球的分布式缓存集群，其实质是通过智能 DNS 判断用户的来源地域及上网线路，为用户选择一个最接近用户地域，以及和用户上网线路相同的服务器节点，因为地域近，且线路相同，所以，可以大幅提升用户浏览网站的体验。

CDN 产生背景之一：BGP 机房虽然可以提升用户体验，但是价格昂贵，对于用户来说，CDN 的诞生可以提供比 BGP 机房更好的体验（让同一地区、同一线路的用户访问和当地同一线路的网站），BGP 机房和普通机房有将近 5 ~ 10 倍的价格差。CDN 多使用单线的机房，根据用户的线路及位置，为用户选择靠近用户的位置，以及相同的运营商线路，不但提升了用户体验，价格也降了下来。

CDN 的价值：

❑ 为架设网站的企业省钱。

❑ 提升企业网站的用户访问体验（相同线路、相同地域、内存访问）。

❑ 可以阻挡大部分流量攻击，例如：DDOS 攻击。

经常有朋友问我，日 100 万 PV 的架构如何设计？对于这样的问题，下面简单为大家提供解答思路：首先应尽量考虑把网站数据放到 CDN 中缓存，这样计算在网站中的总流量和总访问量后减去 CDN 的访问流量，剩下的访问量规模需要的架构才是我们需要设计和考虑的。一个好的网站架构设计，访问量应尽量都交给 CDN。

## 12.10.2 CDN 的特点

CDN 就是一个具备根据用户区域和线路智能调度的分布式内存缓存集群。其特点如下：

❑ 通过服务器内存缓存网站数据，提高了企业站点（尤其是含有大量图片、视频等的站点）的访问速度，并大大提高了企业站点的稳定性（省钱且提升用户体验）。

❑ 用户根据智能 DNS 技术自动选择最适合的 Cache 服务器，降低了不同运营商之间互联瓶颈造成的影响，实现了跨运营商的网络加速，保证不同网络中的用户都能得到良好的访问质量。

❑ 加快了访问速度，减少了原站点的带宽。

❑ 用户访问时从服务器的内存中读取数据，分担了网络流量，同时减轻了原站点负载的压力等。

❑ 使用 CDN 可以分担源站的网络流量，同时还可以减轻原站点的负载压力，并降低黑客入侵及各种 DDoS 攻击对网站的影响，从而保证网站有较好的服务质量。

下面是通过 curl 命令访问 163 网站的 header 信息，可以看到 163 网站的首页就使用了

CDN 进行加速。

```
[oldboy@student ~]$ curl -I www.163.com
HTTP/1.1 200 OK
Expires: Sat, 25 Feb 2012 02:58:57 GMT
Date: Sat, 25 Feb 2012 02:57:37 GMT
Server: nginx
Content-Type: text/html; charset=GBK
Transfer-Encoding: chunked
Vary: Accept-Encoding
Cache-Control: max-age=80
Vary: User-Agent
Vary: Accept
Age: 60
X-Via: 1.1 zb200:80 (Cdn Cache Server V2.0), 1.1 jxq206:8107 (Cdn Cache Server V2.0)
Connection: keep-alive
```

上面的“(Cdn Cache Server V2.0)”表示 163 首页使用了 CDN 加速。

### 12.10.3 企业使用 CDN 的基本要求

首先要说的是，不是所有的网站都可以一上来就能用 CDN 的。要加速的业务数据应该存在独立的域名，例如：`img1-4.etiantian.org/video1-4.etiantian.org`，业务内容图片、附件、JS、CSS 等静态元素，这样的静态网站域名才可以使用 CDN。

下面来看一个 DNS 解析范例。DNS 服务器加速前的 A 记录如下：

```
;A records
img.etiantian.org    IN A          124.106.0.21 (企业服务器的 IP)
```

删除上面的记录，命令如下：

```
img.etiantian.org    IN A          124.106.0.21 (服务器的 IP)
```

然后，做下面的别名解析：

```
; CNAME records
img1.etiantian.org    IN CNAME bbs
img.etiantian.org     3M IN CNAME  img.etiantian.org.cachecn.com.
```

这个 `img.etiantian.org.cachecn.com` 地址必须是事先由 CDN 公司配置好的 CDN 公司的域名。国内较大的 CDN 提供商为网宿、蓝讯、快网。

## 12.11 Nginx 程序架构优化


解耦是开发人员中流行的一个名词，简单地讲就是把一堆程序代码按照业务用途分开，然后提供服务，例如：注册登录、上传、下载、浏览列表、商品内容页面、订单支付等都应

该是独立的程序服务，只不过在客户端看来是一个整体而已。如果中小公司做不到上述细致的解耦，起码也要让下面的几个程序模块独立。

- 网页页面服务。
- 图片附件及下载服务。
- 上传图片服务。

上述三者的功能应尽量分离。分离的最佳方式是分别使用独立的服务器（需要改动程序），如果程序实在不易更改，那么次选方案是在前端负载均衡器 Haproxy/Nginx 上，根据 URI（例如目录或扩展名）过滤请求，然后抛给后面对应的服务器。

例如：根据扩展名分发，请求 `http://www.etiantian.org/a/b.jpg` 的就应抛给图片服务器（独立的静态服务器最适合使用 CDN）；根据 URL 路径分发，请求 `http://www.etiantian.org/upload/index.php` 的就应抛给上传服务器。不符合上面两个要求的，就默认抛给 Web 服务器。

 **说明** 可以部署 3 台服务器，人为分布请求服务器。当然了，这适合并发比较高、服务器较多的情况。程序架构分离了，效率、安全性都会提高很多。

## 12.12 使用普通用户启动 Nginx（监牢模式）

### 12.12.1 为什么要让 Nginx 服务使用普通用户

默认情况下，Nginx 的 Master 进程使用的是 root 用户，worker 进程使用的是 Nginx 指定的普通用户，使用 root 用户运行 Nginx 的 Master 进程有两个最大的问题：

- 管理权限必须是 root，这就使得最小化分配权限原则遇到难题。
- 使用 root 运行 Nginx 服务，一旦网站出现漏洞，用户就可以很容易地获得服务器的 root 权限。

因此，我想出了一种不用为开发人员，甚至普通运维人员赋予管理员权限，就可以很好地管理 Nginx 服务的方法，具体内容将在下节为大家讲解。

### 12.12.2 给 Nginx 服务降权的解决方案

解决方案如下：

- 给 Nginx 服务降权，用 inca 用户运行 Nginx 服务，给开发及运维设置普通账号，只要与 inca 同组即可管理 Nginx，该方案解决了 Nginx 的管理问题，防止 root 分配权限过大。
  - 开发人员使用普通账户即可管理 Nginx 服务及站点下的程序和日志。
  - 采取项目负责制度，即谁负责项目维护，出了问题就是谁负责。
- 很多公司的开发和运维为 root 权限争得不可开交，甚至大打出手。

参考资料：到底要不要给开发人员管理服务器的权限？（<http://down.51cto.com/data/844517>）。

### 12.12.3 给 Nginx 服务降权实战

本优化属于架构优化（同样适合于其他软件），通过 Nginx 启动命令的 `-c` 参数指定不同的 Nginx 配置文件，可以同时启动多个实例，并使用普通的用户运行服务。

Nginx 安装后的启动命令路径为“`/application/nginx/sbin/nginx`”，可以通过加 `-h` 参数查看相关参数的用法，命令及结果如下：

```
[root@oldboy ~]# /application/nginx/sbin/nginx -h
nginx version: nginx/1.3.4
Usage: nginx [-?hvVtq] [-s signal] [-c filename] [-p prefix] [-g directives]
Options:
  -?, -h           : this help
  -v               : show version and exit#<== 显示 Nginx 版本号后退出
  -V               : show version and configure options then exit#<== 显示 Nginx 版本
                    号和配置选项后退出
  -t               : test configuration and exit#<== 测试配置文件是否正确，在运行时需要重
                    新加载配置。此命令非常重要，用来检测所修改的配置文件是否有语法错误
  -q               : suppress non-error messages during configuration testing
  -s signal        : send signal to a master process: stop, quit, reopen, reload
                    #<== 发送信号给一个 master 进程，这里的 reload 参数很重要，是优雅重启 Nginx 的参数，类似
                    Apache 的 graceful 参数
  -c filename      : set configuration file (default: conf/nginx.conf)#<== 使用指定
                    的配置文件而不是 conf 目录下的 nginx.conf
```

这里我们就是通过 `nginx -c` 路径的功能来实现运行多实例 Nginx

较常用的方法是使服务运行在指定用户的根目录下面，这样相对比较安全，同时有利于批量业务部署和上线。

配置普通用户启动 Nginx 的过程如下。

（1）添加用户并创建相关的目录和文件，操作如下：

```
[root@www ~]# useradd inca
[root@www ~]# su - inca
[inca@www ~]$ pwd
/home/inca
[inca@www ~]$ mkdir conf logs www
[inca@www ~]$ cp /application/nginx/conf/mime.types ~/conf/
[inca@www ~]$ echo inca >www/index.html
```

（2）配置 Nginx 配置文件。配置后的查看命令如下：

```
[inca@www ~]$ cat conf/nginx.conf
worker_processes 4;
worker_cpu_affinity 0001 0010 0100 1000;
worker_rlimit_nofile 65535;
```

```

error_log /home/inca/logs/error.log;
user inca inca;
pid /home/inca/logs/nginx.pid;
events {
    use epoll;
    worker_connections 10240;
}
http {
    include mime.types;
    default_type application/octet-stream;
    sendfile on;
    keepalive_timeout 65;

    log_format main '$remote_addr - $remote_user [$time_local] "$request" '
        '$status $body_bytes_sent "$http_referer" '
        '"$http_user_agent" "$http_x_forwarded_for"';

    #web.fei fa daolian.....
    server {
        listen 8080;
        server_name www.etiantian.org;
        root /home/inca/www;
        location / {
            index index.php index.html index.htm;
        }
        access_log /home/inca/logs/web_blog_access.log main;
    }
}

[inca@www ~]$ LANG=en
[inca@www ~]$ tree
.
|-- conf
|   |-- mime.types
|   |-- nginx.conf
|-- logs
|-- www
    |-- index.html
3 directories, 3 files

```

说明如下：

- ❑ 所有参数的值，带路径的都要改成 /home/inca。
- ❑ 特权用户 root 使用的是 80 端口，改为普通用户使用的端口，在 1024 以上，这里为 8080。

(3) 启动 Nginx，命令如下：

```

[inca@www ~]$ ps -ef|grep nginx|grep -v grep
[inca@www ~]$ /application/nginx/sbin/nginx -c /home/inca/conf/nginx.conf &>/dev/
null &

```



```
[1] 4736
[inca@www ~]$ ps -ef|grep nginx|grep -v grep
inca      4738      1  0 13:02 ?          00:00:00 nginx: master process /appli-
cation/nginx/sbin/nginx -c /home/inca/conf/nginx.conf #<== 主进程也是 inca 用户了
inca      4739  4738  0 13:02 ?          00:00:00 nginx: worker process
inca      4740  4738  0 13:02 ?          00:00:00 nginx: worker process
inca      4741  4738  0 13:02 ?          00:00:00 nginx: worker process
inca      4742  4738  0 13:02 ?          00:00:00 nginx: worker process
[inca@www ~]$ ss -lntup|grep nginx
tcp LISTEN 0 128 *:8080      *:~ users: (("nginx",4738,5), ("nginx",4739,5), ("n
ginx",4740,5), ("nginx",4741,5), ("nginx",4742,5))
# 特别提示：此处启动 Nginx，如果不定向到空会显示一些提示，不是错误，可以通过加 &>/dev/null 定
向到空，从而忽略不见。
[inca@www ~]$ killall nginx
[inca@www ~]$ killall nginx
nginx: no process killed
[inca@www ~]$ /application/nginx/sbin/nginx -c /home/inca/conf/nginx.conf
nginx: [alert] could not open error log file: open() "/application/nginx-1.6.2/
logs/error.log" failed (13: Permission denied)
2015/04/26 13:06:44 [warn] 4762#0: the "user" directive makes sense only if the master
process runs with super-user privileges, ignored in /home/inca/conf/nginx.conf:5
```

#### (4) 查看访问，命令如下：

```
[inca@www ~]$ lsof -i :8080
COMMAND  PID USER  FD   TYPE DEVICE SIZE/OFF NODE NAME
nginx    4764 inca   5u    IPv4 333019      0t0  TCP *:webcache (LISTEN)
nginx    4765 inca   5u    IPv4 333019      0t0  TCP *:webcache (LISTEN)
nginx    4766 inca   5u    IPv4 333019      0t0  TCP *:webcache (LISTEN)
nginx    4767 inca   5u    IPv4 333019      0t0  TCP *:webcache (LISTEN)
nginx    4768 inca   5u    IPv4 333019      0t0  TCP *:webcache (LISTEN)
```

配置好运行解析，浏览器带端口访问结果如图 12-21 所示。

#### (5) 解决普通端口非 80 提供服务的问题。

用负载均衡器解决 Web 服务非 80 端口的转换问题，负  
载均衡器可为 Haproxy、Nginx、F5 等。

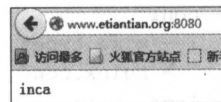


图 12-21 成功展示页面内容

本解决方案的优点如下：

- ❑ 给 Nginx 服务降权，让网站更安全。
- ❑ 按用户设置站点权限，使站点更独立（无需虚拟化隔离）。
- ❑ 开发不需要用 root 即可完整管理服务及站点。
- ❑ 可实现责任划分，即网络问题属于运维的责任，网站打不开就是开发的责任，或者两者共同承担。

## 12.13 控制 Nginx 并发连接数量

ngx\_http\_limit\_conn\_module 这个模块用于限制每个定义的 key 值的连接数，特别是单

IP 的连接数。

不是所有的连接数都会被计数。一个符合计数要求的连接是整个请求头已经被读取的连接。

控制 Nginx 并发连接数量参数的说明如下。

### 1) limit\_conn\_zone 参数

语法: `limit_conn_zone key zone=name:size;`

上下文: `http`

用于设置共享内存区域, `key` 可以是字符串, Nginx 自带变量或前两个组合, 如 `$binary_remote_addr`、`$server_name`。`name` 为内存区域的名称, `size` 为内存区域的大小。

### 2) limit\_conn 参数

语法: `limit_conn zone number;`

上下文: `http`、`server`、`location`

用于指定 `key` 设置的最大连接数。当超过最大连接数时, 服务器会返回 503 (Service Temporarily Unavailable) 错误。

## 1. 限制单 IP 并发连接数

Nginx 的配置文件如下:

```
[root@oldboy ~]# cat /application/nginx/conf/nginx.conf
worker_processes 1;
events {
    worker_connections 1024;
}
http {
    include mime.types;
    default_type application/octet-stream;
    sendfile on;
    keepalive_timeout 65;

    limit_conn_zone $binary_remote_addr zone=addr:10m;

    server {
        listen 80;
        server_name www.etiantian.org;
        location / {
            root html;
            index index.html index.htm;
            limit_conn addr 1; #<== 限制单 IP 的并发连接为 1
        }
    }
}
```

在客户端 10.0.0.5 使用 Apache 的 `ab` 测试工具进行测试。

测试 1: 模拟并发连接 1, 访问 10 次服务器, 即执行 `ab -c 1 -n 10 http://10.0.0.3/` 进行测试。



**注意** -c 为并发数，-n 为请求总数，10.0.0.3 为 Nginx 的 IP 地址。

测试过程中查看 Nginx 的访问日志，结果如下：

```
[root@oldboy ~]# tailf /application/nginx/logs/access.log
10.0.0.5 - - [14/Sep/2015:11:50:31 +0800] "GET / HTTP/1.0" 200 612 "-" "ApacheBench/2.3"
10.0.0.5 - - [14/Sep/2015:11:50:31 +0800] "GET / HTTP/1.0" 200 612 "-" "ApacheBench/2.3"
10.0.0.5 - - [14/Sep/2015:11:50:31 +0800] "GET / HTTP/1.0" 200 612 "-" "ApacheBench/2.3"
10.0.0.5 - - [14/Sep/2015:11:50:31 +0800] "GET / HTTP/1.0" 200 612 "-" "ApacheBench/2.3"
10.0.0.5 - - [14/Sep/2015:11:50:31 +0800] "GET / HTTP/1.0" 200 612 "-" "ApacheBench/2.3"
10.0.0.5 - - [14/Sep/2015:11:50:31 +0800] "GET / HTTP/1.0" 200 612 "-" "ApacheBench/2.3"
10.0.0.5 - - [14/Sep/2015:11:50:31 +0800] "GET / HTTP/1.0" 200 612 "-" "ApacheBench/2.3"
10.0.0.5 - - [14/Sep/2015:11:50:31 +0800] "GET / HTTP/1.0" 200 612 "-" "ApacheBench/2.3"
10.0.0.5 - - [14/Sep/2015:11:50:31 +0800] "GET / HTTP/1.0" 200 612 "-" "ApacheBench/2.3"
10.0.0.5 - - [14/Sep/2015:11:50:31 +0800] "GET / HTTP/1.0" 200 612 "-" "ApacheBench/2.3"
```

根据上述日志可以看出当并发为 1 时，返回值都是 200，即访问正常。

测试 2：模拟并发连接 2，访问 10 次服务器，即执行 `ab -c 2 -n 10 http://10.0.0.3/` 进行测试。

测试过程中查看 Nginx 的访问日志，结果如下：

```
10.0.0.5 - - [14/Sep/2015:11:52:15 +0800] "GET / HTTP/1.0" 200 612 "-" "ApacheBench/2.3"
10.0.0.5 - - [14/Sep/2015:11:52:15 +0800] "GET / HTTP/1.0" 503 212 "-" "ApacheBench/2.3"
10.0.0.5 - - [14/Sep/2015:11:52:15 +0800] "GET / HTTP/1.0" 200 612 "-" "ApacheBench/2.3"
10.0.0.5 - - [14/Sep/2015:11:52:15 +0800] "GET / HTTP/1.0" 200 612 "-" "ApacheBench/2.3"
10.0.0.5 - - [14/Sep/2015:11:52:15 +0800] "GET / HTTP/1.0" 503 212 "-" "ApacheBench/2.3"
10.0.0.5 - - [14/Sep/2015:11:52:15 +0800] "GET / HTTP/1.0" 200 612 "-" "ApacheBench/2.3"
10.0.0.5 - - [14/Sep/2015:11:52:15 +0800] "GET / HTTP/1.0" 503 212 "-" "ApacheBench/2.3"
10.0.0.5 - - [14/Sep/2015:11:52:15 +0800] "GET / HTTP/1.0" 200 612 "-" "ApacheBench/2.3"
10.0.0.5 - - [14/Sep/2015:11:52:15 +0800] "GET / HTTP/1.0" 503 212 "-" "ApacheBench/2.3"
10.0.0.5 - - [14/Sep/2015:11:52:15 +0800] "GET / HTTP/1.0" 200 612 "-" "ApacheBench/2.3"
10.0.0.5 - - [14/Sep/2015:11:52:15 +0800] "GET / HTTP/1.0" 503 212 "-" "ApacheBench/2.3"
```

可以看到状态码 200 与 503 间隔 1:1 出现，即 Nginx 已经做了并发连接限制，对超过限制的请求返回 503。

测试 3：模拟并发连接 3，访问 10 次服务器，即执行 `ab -c3 -n 10 http://10.0.0.3/` 进行测试。

测试过程中查看 Nginx 的访问日志，结果如下：

```
10.0.0.5 - - [14/Sep/2015:11:53:59 +0800] "GET / HTTP/1.0" 200 612 "-" "ApacheBench/2.3"
10.0.0.5 - - [14/Sep/2015:11:53:59 +0800] "GET / HTTP/1.0" 200 612 "-" "ApacheBench/2.3"
10.0.0.5 - - [14/Sep/2015:11:53:59 +0800] "GET / HTTP/1.0" 503 212 "-" "ApacheBench/2.3"
10.0.0.5 - - [14/Sep/2015:11:53:59 +0800] "GET / HTTP/1.0" 503 212 "-" "ApacheBench/2.3"
10.0.0.5 - - [14/Sep/2015:11:53:59 +0800] "GET / HTTP/1.0" 200 612 "-" "ApacheBench/2.3"
10.0.0.5 - - [14/Sep/2015:11:53:59 +0800] "GET / HTTP/1.0" 503 212 "-" "ApacheBench/2.3"
10.0.0.5 - - [14/Sep/2015:11:53:59 +0800] "GET / HTTP/1.0" 503 212 "-" "ApacheBench/2.3"
```

```
10.0.0.5 - - [14/Sep/2015:11:53:59 +0800] "GET / HTTP/1.0" 200 612 "-" "ApacheBench/2.3"
10.0.0.5 - - [14/Sep/2015:11:53:59 +0800] "GET / HTTP/1.0" 200 612 "-" "ApacheBench/2.3"
10.0.0.5 - - [14/Sep/2015:11:53:59 +0800] "GET / HTTP/1.0" 503 212 "-" "ApacheBench/2.3"
```

由于采用的样本较小，所以出现的次数并不均衡。但看其中间数据，200 与 503 出现的次数为 1:2，即 Nginx 已经做了并发连接限制，对超过限制的请求返回 503。

以上功能的应用场景之一是用于服务器下载，命令如下：

```
location /download/ {
    limit_conn addr 1;
}
```

上面的命令限制访问 download 下载目录的连接数，该连接数为 1。


## 2. 限制虚拟主机总连接数

不仅可以限制单 IP 的并发连接数，还可以限制虚拟主机的总连接数，甚至可以对两者同时进行限制。Nginx 的配置文件如下：

```
[root@oldboy ~]# cat /application/nginx/conf/nginx.conf
worker_processes 1;
events {
    worker_connections 1024;
}
http {
    include mime.types;
    default_type application/octet-stream;
    sendfile on;
    keepalive_timeout 65;

    limit_conn_zone $binary_remote_addr zone=addr:10m;
    limit_conn_zone $server_name zone=perserver:10m;

    server {
        listen 80;
        server_name www.etiantian.org;
        location / {
            root html;
            index index.html index.htm;
            #limit_conn addr 1;
            limit_conn perserver 2; #<== 设置虚拟主机连接数为 2
        }
    }
}
```

 **提示** Nginx 的内部变量列表见官网 <http://nginx.org/en/docs/varindex.html>。

测试：执行 `ab -c 5 -n 1000 http://10.0.0.3/` 进行测试，即并发连接数为 5，访问 1000 次。

统计日志中 200 与 503 出现的次数（测试前清空日志，测完将日志复制到 root 根目录），代码如下：

```
[root@oldboy ~]# grep -c 200 access.log
634
[root@oldboy ~]# grep -c 503 access.log
366
```

结论：出现的次数近似为 2:1。

至此，Nginx 限制连接数的应用实践就讲解完毕了。

更多内容可参考：[http://nginx.org/en/docs/http/nginx\\_http\\_limit\\_conn\\_module.html](http://nginx.org/en/docs/http/nginx_http_limit_conn_module.html)。

## 12.14 控制客户端请求 Nginx 的速率

ngx\_http\_limit\_req\_module 模块用于限制每个 IP 访问每个定义 key 的请求速率。limit\_req\_zone 参数说明如下。

语法：limit\_req\_zone key zone=name:size rate=rate;

上下文：http

用于设置共享内存区域，key 可以是字符串，Nginx 自带变量或前两个组合，如 \$binary\_remote\_addr。name 为内存区域的名称，size 为内存区域的大小，rate 为速率，单位为 r/s，每秒一个请求。

limit\_req 参数说明如下。

语法：limit\_req zone=name [burst=number] [nodelay];

上下文：http、server、location

这里运用了令牌桶原理，burst=num，一共有 num 块令牌，令牌发完后，多出来的那些请求就会返回 503。

换句话说，一个银行，只有一个营业员，银行很小，等候室只有 5 个人的位置。因此，营业员一个时刻只能为一个人提供服务，剩下的不超过 5 个人可以在银行内等待，超出的人不提供服务，直接返回 503。

nodelay 默认在不超过 burst 值的前提下会排队等待处理，如果使用此参数，就会处理完 num+1 次请求，剩余的请求都视为超时，返回 503。

用于测试的 Nginx 配置文件如下：

```
[root@oldboy ~]# cat /application/nginx/conf/nginx.conf
worker_processes 1;
events {
    worker_connections 1024;
}
http {
```

```

include      mime.types;
default_type application/octet-stream;
sendfile     on;
keepalive_timeout 65;
limit_req_zone $binary_remote_addr zone=one:10m rate=1r/s;
#<== 以请求的客户端 IP 作为 key 值，内存区域命名为 one，分配 10MB 内存空间，访问速率限制为 1
秒 1 次请求 (request)

server {
    listen      80;
    server_name www.etiantian.org;
    location / {
        root    html;
        index   index.html index.htm;
        limit_req zone=one burst=5;
#<== 使用前面定义的名为 one 的内存空间，队列为 5，即可以有 5 个请求排队等待
    }
}

```

测试 1：执行 `ab -c 4 -n 1000 http://10.0.0.3/` 和 `ab -c 5 -n 1000 http://10.0.0.3/` 进行测试，命令及结果如下。

```

[root@oldboy ~]# tailf /application/nginx/logs/access.log
10.0.0.5 - - [14/Sep/2015:13:50:19 +0800] "GET / HTTP/1.0" 200 612 "-" "ApacheBench/2.3"
10.0.0.5 - - [14/Sep/2015:13:50:20 +0800] "GET / HTTP/1.0" 200 612 "-" "ApacheBench/2.3"
10.0.0.5 - - [14/Sep/2015:13:50:21 +0800] "GET / HTTP/1.0" 200 612 "-" "ApacheBench/2.3"
10.0.0.5 - - [14/Sep/2015:13:50:22 +0800] "GET / HTTP/1.0" 200 612 "-" "ApacheBench/2.3"
10.0.0.5 - - [14/Sep/2015:13:50:23 +0800] "GET / HTTP/1.0" 200 612 "-" "ApacheBench/2.3"
10.0.0.5 - - [14/Sep/2015:13:50:24 +0800] "GET / HTTP/1.0" 200 612 "-" "ApacheBench/2.3"
10.0.0.5 - - [14/Sep/2015:13:50:25 +0800] "GET / HTTP/1.0" 200 612 "-" "ApacheBench/2.3"
.....

```

可以发现，访问日志中的时间和请求是 1 秒钟 1 条请求，证明配置生效。

测试 2：执行 `ab -c 6 -n 1000 http://10.0.0.3/` 进行测试。

统计日志中的 200 与 503 出现的次数（测试前已清空日志，测完将日志复制到 root 根目录）如下：

```

[root@oldboy ~]# grep -c 200 access.log
6
[root@oldboy ~]# grep -c 503 access.log
994

[root@oldboy ~]# more access.log
10.0.0.5 - - [14/Sep/2015:13:48:42 +0800] "GET / HTTP/1.0" 200 612 "-" "ApacheBench/2.3"
10.0.0.5 - - [14/Sep/2015:13:48:42 +0800] "GET / HTTP/1.0" 503 212 "-" "ApacheBench/2.3"
10.0.0.5 - - [14/Sep/2015:13:48:42 +0800] "GET / HTTP/1.0" 503 212 "-" "ApacheBench/2.3"
10.0.0.5 - - [14/Sep/2015:13:48:42 +0800] "GET / HTTP/1.0" 503 212 "-" "ApacheBench/2.3"
10.0.0.5 - - [14/Sep/2015:13:48:42 +0800] "GET / HTTP/1.0" 503 212 "-" "ApacheBench/2.3"
10.0.0.5 - - [14/Sep/2015:13:48:42 +0800] "GET / HTTP/1.0" 503 212 "-" "ApacheBench/2.3"

```

.....省略若干.....

.....省略若干.....

```
10.0.0.5 - - [14/Sep/2015:13:48:42 +0800] "GET / HTTP/1.0" 503 212 "-" "ApacheBench/2.3"
10.0.0.5 - - [14/Sep/2015:13:48:42 +0800] "GET / HTTP/1.0" 503 212 "-" "ApacheBench/2.3"
10.0.0.5 - - [14/Sep/2015:13:48:42 +0800] "GET / HTTP/1.0" 503 212 "-" "ApacheBench/2.3"
10.0.0.5 - - [14/Sep/2015:13:48:43 +0800] "GET / HTTP/1.0" 200 612 "-" "ApacheBench/2.3"
10.0.0.5 - - [14/Sep/2015:13:48:44 +0800] "GET / HTTP/1.0" 200 612 "-" "ApacheBench/2.3"
10.0.0.5 - - [14/Sep/2015:13:48:45 +0800] "GET / HTTP/1.0" 200 612 "-" "ApacheBench/2.3"
10.0.0.5 - - [14/Sep/2015:13:48:46 +0800] "GET / HTTP/1.0" 200 612 "-" "ApacheBench/2.3"
10.0.0.5 - - [14/Sep/2015:13:48:47 +0800] "GET / HTTP/1.0" 200 612 "-" "ApacheBench/2.3"
```

通过过滤排重及部分日志，可以看到第一个请求返回 200，为正常处理，剩余的 994 次超过限制的请求在 1 秒内执行完成，但是都返回了 503。

具体过程原理为：Nginx 在第 1 秒先处理第一个请求，同时接下来的 5 个请求等待排队，剩下的所有（994 次）请求返回 503。接着第 2 秒到第 6 秒处理等待的 5 个请求。

更多内容可参考：[http://nginx.org/en/docs/http/ngx\\_http\\_limit\\_req\\_module.html](http://nginx.org/en/docs/http/ngx_http_limit_req_module.html)。

## 12.15 小结

本章重点回顾：

- (1) 安全优化：隐藏 Nginx 软件名及版本号。
- (2) 性能加安全优化：连接超时参数及 FastCGI 相关参数调优。
- (3) 性能优化：gzip 压缩功能及调试查看方法。
- (4) 性能优化：expires 缓存功能及调试查看方法。
- (5) 安全优化：集群中各角色服务站点目录权限控制策略。
- (6) 安全优化：站点目录下所有的文件和目录访问控制。
- (7) 性能加安全优化：robots.txt 协议及防爬虫优化解决方案。
- (8) 性能加安全优化：静态资源防盗链解决方案。
- (9) 用户体验优化：错误页面优雅显示方法。
- (10) 安全优化：限制 HTTP 请求方法。
- (11) 性能加安全优化：CDN 加速知识。
- (12) 安全优化：监牢模式运行 Nginx 方案策略。
- (13) 性能加安全优化：Nginx 并发连接数及请求速率控制。



## 游戏运维的思考

### 作者简介

马亮，原搜狐畅游端游研发，端游、手游、运维开发主管，现腾讯云自身游戏资深售前架构师、游戏云高级产品经理。

### 13.1 游戏运维最关键的几件事

个人认为，游戏运维的核心指标主要包含 4 个方面：安全、稳定、高效、成本节约。其中，安全最为优先，任何因安全问题造成的玩家数据丢失或版本泄露都是致命的打击；业务的稳定次之；高效的前提是稳定，没有稳定的基础再高效都是无用功；成本问题应该是在满足上述前提下最终的考量指标，切不可为了节约成本而影响了上述的三个指标。

#### 13.1.1 安全

安全游戏运维领域主要关注版本安全、数据安全、网络安全、主机安全和业务安全几个维度，其中版本安全 and 数据安全需要我们对数据和版本的每个环节都做到全路径掌控，要冗余再冗余、备份再备份，关键路径用人必须谨慎。网络、主机及业务安全需要我们对当前面临的安全攻击行为提前做好检测和防御，严格控制入口，对系统的选型应尽可能稳重些，尽可能降低操作的权限。

##### 1. 版本安全

无论游戏发展到哪个阶段，私服都会给官服的正常运营带来巨大的负面影响，无论是当

年私服兴起的端游时代，还是现在私服屡禁不止的页游和手游时代，各大游戏公司都会想尽各种办法，采用各种安全措施来防范版本泄露。

从全路径掌控上来说，需要管控如下几个方面。

- ❑ 游戏版本的全内网研发环境管控。
- ❑ 游戏版本从内网到外网迁出的流程通道的管控：数据默认不能直接流出，物理上做了隔离，流出环节中间要有转换，而且流出环节要有审核关键的操作人进行控制。
- ❑ 游戏版本到外网的操作环境管控：对文件系统做软加密 PGP、对操作版本的通道做堡垒机及操作指令的限制、对版本部署的环境做三层的安全管控。
- ❑ 对游戏运行方式的管控：对游戏服的启动要有认证环，以控制启动环节。
- ❑ 尽最大范围地管控版本的全路径环境。
- ❑ 提供最小范围的允许度，对涉及人员的环节，权限拆分得越细越好（系统、应用运维和数据库运维），降低一个人对整体版本的可控度，而且对涉及关键环节的人物的选择需要绝对的谨慎和重视。

另外，每个环节都需要有冗余的架构，从版本的内网环境到通道环境再到线上环境都要做到硬件的冗余（硬件上是双节点之间进行定期同步、磁盘会做 RAID、网络设备上会做双链路冗余 bond 架构），系统操作层面的冗余（服务本身也会做定期同步和冗余），任何的单点风险都应提前杜绝，之前游戏日志会有定期的保留。之前发生日志服务单节点的故障会造成业务数据的丢失或难以恢复，这样造成的直接影响就是历史数据将会难以分析，一些被盗召回的行为将难以定位，等等。采用了冗余的架构，单点的风险就能够降低很多，架构上的冗余能保证业务的整体流程不会因为硬件或软件的故障带来风险，与此同时我们还需要对其中的数据操作环节提前做好备份，确保每个环节中的数据都有冗余和留存，整体的数据环节也需要按照不同的保留环节做好保留的策略（见图 13-1）：

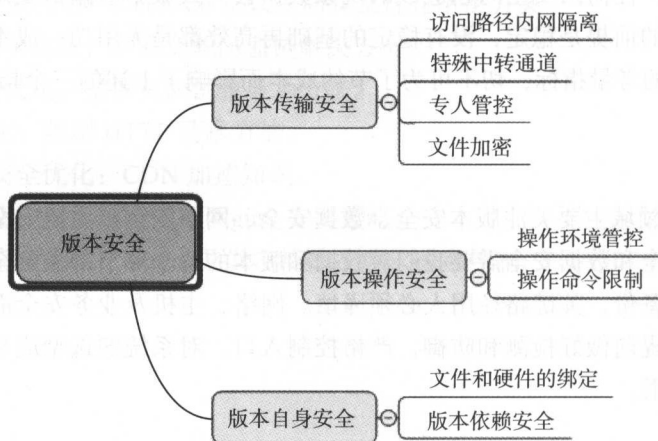


图 13-1 版本安全示意图

## 2. 数据安全

游戏数据是玩家的命根子，处理的时候千万马虎不得。从数据的生成到数据的传输，再到数据的保留、归档，每个环节都需要把可能影响数据完整性、数据一致性的因素提前考虑进来，提前就每个环节做好冗余和管控。

首先，数据怎么做备份、数据库怎么做冗余都不过分，任何一次的回档对一款游戏而言都是很大甚至致命的伤害。数据通常是存储到数据库中或文件系统中，那么数据存储的环节，会有硬件层、系统层和应用层，哪一层出现问题对数据的可靠性都会有安全隐患，目前的技术都很成熟了，无论是 NoSQL 还 SQL 基于系统和软件层都做了冗余，就连硬件的问题也做了冗余的方案。

其次，从数据的生命周期的环节来看，整个游戏数据的构建、更新、备份和清理环节都存在失败的风险和不可控度，就拿备份环节来说吧，你怎么备份，备份到哪里，备份完怎么校验，这些都需要做好每个环节的检测和验收，有的业务备份会放到集中的存储地区，传输环节需要做验证，备份完成了对目标存储空间的大小也要做检测，至少不要发生传过去的资料丢失的问题。

总之，做数据的工作一定要细致，做任何操作之前都要先备份一份再说，哪怕用不到也没关系，数据操作中本来就存在很多的不可控性，数据更新环节、流程至关重要，直接跳流程会对业务库造成不可挽回的损失。一定要先做好验证，数据操作脚本内容的完整性即使确认 3 次都不为过。提前拿备份来做验证，运行通了再说。真正更新前也一定要先进行备份，再进行操作，因为你不知道在操作环节中会遇到什么样的问题，有可能操作一半系统就崩溃了，软件发生崩溃也有可能，我们在这里能多一些就多做一些，当然所做的都是必需的环节，做多了反而会添乱。很多时候解决问题表面上添加一个环节就是管控，其实也增加了一层的风险。如果每次都这么做，操作环节将会越来越复杂，人为的风险更不可控，从底层来做反而更直接，例如分布式存储，实现对数据的 3 份分散的备份对存储的安全就做得很彻底。

图 13-2 为数据安全示意图：

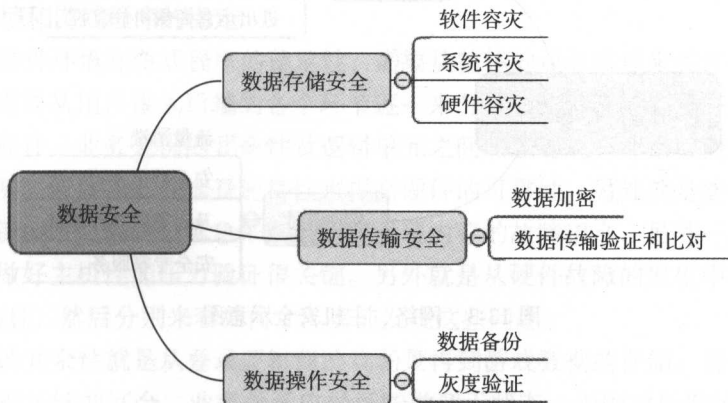


图 13-2 数据安全示意图

3. 网络、主机及业务安全

网络、主机及业务安全是游戏运维安全的重中之重，没有问题的时候一切都好说，一旦出现问题了就有可能是致命的打击，例如 DDoS 攻击，平时可能都没事，一旦在游戏的关键阶段出现了问题，将会是非常麻烦的事情，这里有很多工作需要尽快去做。目前的公有云，或多或少都有安全设备来解决 DDoS 攻击的问题。之前没有的时候要么自己购买，要么使用 IDC 的，对于这方面的分析能力和定位能力需要投入人力来把关，如果要用 IDC，则需要测试下是否真的有效，提前模拟 DDoS 攻击以检测 IDC 是否真的能防得住，如果防不住要么就迁移到公有云上要么就购买相关产品。如果都不想选择，那么就要找运营商对一些协议做控制，假如游戏使用的是 TCP 协议，就要限制 UDP 协议，把 IP 的访问规则做得谨慎些。另外就是找一些靠谱的 IDC，将出口带宽设置得大一些。有时可能不是你的业务受到了攻击，而是同一出口其他的业务受到了攻击从而影响了你的业务。把业务分散放置，不要都放在一个地方，否则一个地方出了问题整个业务都会挂掉。另外就是和 IDC 做好联动，用一些流量分析的软件关注下攻击源或目标源，如果攻击源有一定的规律就让 IDC 协助来做限制，只要先把安全攻击想明白了再做最简单的控制就容易多了。

当然还会有更多的安全攻击行为，这里能做的就是严格控制访问，使操作行为可控制，如果还是不行就借助第三方服务。对于主机访问尽可能采用中间代理的模式，限制代理的访问模式，严格管控好防火墙，并且对操作方式做严格的审查和操作权限细粒度的控制，定期关注端口的开放情况及漏洞情况，及时打补丁。如有必要则找专家来做安全渗透的测试。一旦出现问题，如果业务架构足够冗余则直接重装；如果实在不行则找专家做好安全分析，找出后门和攻击方式。

图 13-3 为网络、主机安全示意图：

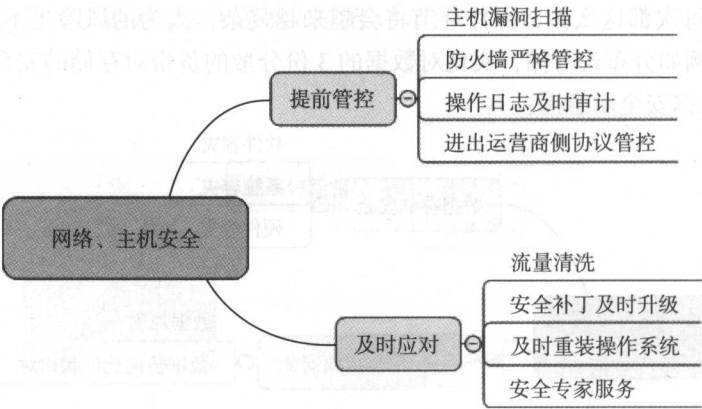


图 13-3 网络、主机安全示意图

13.1.2 稳定

稳定性上需要从我们提供服务的每个关键环节去梳理（网络稳定，硬件环境的稳定，游

戏稳定, 游戏运维的人员稳定), 把握好了每个环节的的稳定, 整体就相对稳定了。那么具体该如何做到呢?

### 1. 网络稳定

先说说网络, 国内网络的复杂程度不言而喻, 从整体的体制架构来看, 管理上的分权, 业务架构运维的标准均不统一, 加上运营商的垄断特性, 对我们而言的不透明和任意性, 给本身就复杂的网络环境增加了更多的不可控性, 我们能做的就是找到标杆的 IDS, 尽可能地对业务资源进行整合管理以提高我们在 IDC 环节的话语权, 让 IDC 的变更对我们透明化, 尽可能地了解运营商的内部, 业务上线前在资源采购上先做好一定的冗余, 之前我们就发现运营商提供的资源和实际采购的资源有误差, 刚开始多买点没关系, 稳定了就退掉, 对可控的地方尽量都做到可控, 尽可能减少不可控的环节, 或者通过第三方云服务来帮我们做到可控, 对于任何一家云服务我们都需要提前做好稳定性的监控和分析, 还要看该公司的经验和沉淀, 该云服务公司是否有抱负, 是否有资源, 是否有足够专业的人在做, 是否能够满足网络架构的多出口和网络设备的双上联容灾, 并且在业务上应尽可能地尝试脱敏。

图 13-4 为网络稳定示意图:

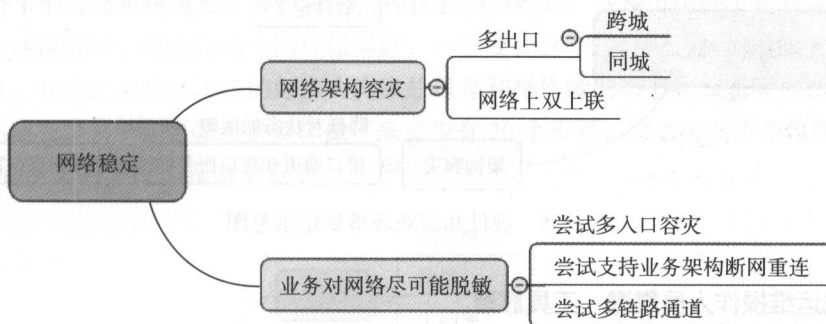


图 13-4 网络稳定示意图

### 2. 硬件环境和游戏程序的稳定

要想保证硬件环境和游戏程序的稳定性, 需要从整体的业务流程来关注每个环节的稳定性, 这里我们需要从用户接入后端的各个环节逐一来分析, 我们需要保障好承载业务的每个物理单元的可靠性、业务架构的冗余性及逻辑单元之间的互通性。那么如何来保障每个环节的硬件稳定性呢? 可以从硬件本身的特性来提高硬件的可靠性。另外就是能够提前发现硬件的故障是很关键的, 之前我们紧急采购的资源发现内存的故障很高, 另外一次就是磁盘出现了故障, 提前做好主机性能压力验证很关键。另外就是从硬件故障的发生中查找规律, 看看 smart 的报错信息, 然后分别来看如何才能提前发现这些问题。

业务架构的冗余性就是从登录逻辑到游戏场景再到游戏数据的存储, 每个环节都需要从业务架构上做到足够的冗余, 业务单元应尽可能做到无状态, 否则对后端的压力就会更大。例如登录环节, 能做网关的就做网关, 一定不能有状态, 任何一处故障对玩家而言就需要重



新连接。逻辑服通常都是有状态的，应尽可能地把数据和逻辑处理拆开，数据处理能做文件或者缓存的存储的方式更好，逻辑和数据层之间的同步节奏应尽可能地快，以降低逻辑服宕机对数据造成的风险。而且还需要关注逻辑单元之间的互通性，这里最核心的就是管道的畅通性和效率。管道的畅通性是指关注管道不堵塞，不断掉，例如，管道的带宽是否充裕，是否存在被争抢的可能性。链路上至少做到双通道冗余（可以从硬件上做绑定或从配置上做自动切换），能够随时切换。另外就是效率和延迟，这里需要从硬件层、系统层、链路层，对每个传输单元的性能做好预估，传输管道是否按照最佳路径来配置的，例如这里我们需要关注到网卡、交换机的硬件性能处理能力，业务接口的处理能力，对消息的传输路径确认是否已经做到最佳路径，能走私网就走私网，如果走公网的话是否会有跨网的问题，或者是否按照最佳的路径来传递。

图 13-5 所示的为硬件和游戏环境稳定示意图：

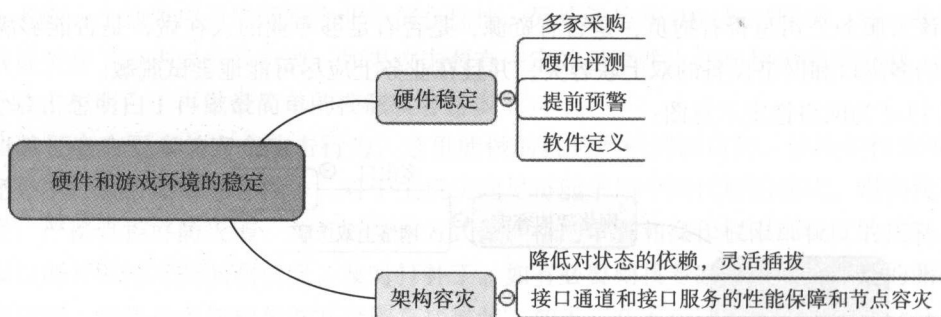


图 13-5 硬件和游戏环境稳定示意图

### 3. 游戏运维操作人员靠谱、工具靠谱

我们需要对每次操作的必要性，每次操作的流程都要有清晰的认知，任何一次没有必要的变更都会带来不稳定的因素，不动最好，或者少做无用功，这属于业务的决策。之前做项目的时候有时做一个决策是为了修复业务，对新增有影响，但对在线无影响，我们是应该马上去操作呢还是在玩家人数少的时候去操作呢。或者偶尔会因为一些故障我们无法连接到主机，但玩家实际上还在体验游戏，那么我们是否应该操作呢。对于这类问题我们都是争取在维护的周期去做，或者在玩家人数少的时候去操作。还有就是之前有遇到过需要马上升级一个脚本之类的问题，其实解决的都是体验性的问题，但是每次维护都需要花费时间，难道这样的升级对玩家的体验会好吗。

在操作环节上，应提前做好操作前的评估，对存在的风险点和操作时间上的预估要相对科学。检查环节必不可少，应尽可能做到每个环节都有验收最好。最好能够做到交叉检查，检查工作多做几次没有坏处，工具上能自动的就不要手动，最大范围地降低对人的判断，每次操作应尽可能地分环境逐步来发布，先在内部的测试环节中更新一次，测试下游戏的基本功能是否正常，核心的消费环节是否有异常，如果没有问题则发布到测试服。测试服根据版

本的大小来做一定时间的验证，如果验证没有问题再全网发布。我们可以根据玩家的属性来对版本包做选择性的变更发布，保证最大范围内降低发布的问题。并且从每一步的发布中发现问题并及时解决和优化，操作环节也要做好最坏的打算，做好回退或分步操作的准备。业务操作环节越少越好，操作步骤越多出现的事故就会越多，能一步做完的不要分成两步，每个环节都会报错，要处理的问题和定位的问题会随着步骤的增加而增加。

操作人员方面，最好是由与项目利益相关的人员的来做。之前我们曾选择值班人员来操作，那个时候值班的人员都是平台性质的岗位，对游戏的架构和配置说明及操作的每一个步骤的理解都不是很深刻，加上平台的岗位与项目利益无关，对人为事故的处罚不严厉，对监控人员的培训不到位等因素，各种问题都爆发出来了，人为事故频率居高不下。后来把业务操作的权限开放给业务的接口人并提供独立的笔记本和弹性的工作时长，整体的人为事故频率才低了些。当然如果工具化做得足够好，操作足够简单，再加上足够的培训和明确的奖惩制度，将业务的操作内容交付给平台的值班人员也是没有问题的。不过有些人做一段时间了就得换掉，换掉那些容易麻痹大意或压根就不适合这个岗位的人，其实人的行为是可以从一点一滴的细节中观察出来的，发生人为事故不是巧合，而是经过一段时间的积累，或者已经形成了习惯。

操作环节的心态也很重要，操作者的精神状态必须要好，昏昏沉沉的工作就容易遗漏步骤或做错，俗话说得好，要想让这个岗位做得好，一定要让这个岗位干得爽，不能过度疲累也不能无所事事，其实之前我们出现的好多人事故都是和操作者的心态和状态有关，加上工作工具化做得不够完善，一次更新维护整体做下来至少有 20 个步骤，怎么可能不犯错呢？

图 13-6 所示的为操作靠谱示意图：

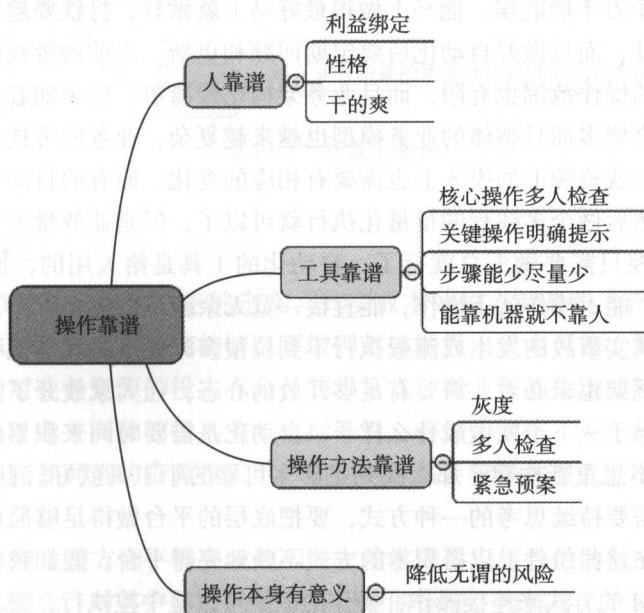


图 13-6 操作靠谱示意图



### 13.1.3 高效

高效的核心就是处理速度快，但快的前提是稳定，千万不能为了快速处理故障导致出现不可控的因素。之前有一次我们的业务线上出现了硬件报警，提示内存有一位校验位错误，为保守起见需要更换一台机器，当时想的就是直接把硬盘切换到另外一台上就好了，更换环节中硬盘更换的时间太长大大出乎我们的意料之外，主要是硬盘更换的操作需要 IDC 的人员来操作，对他们的经验和能力都不是很清楚，更换硬盘操作大大延长了故障的处理时间。如果我们选择切换备机，虽然看似没有这种方式来的简单，但实际确实更为可控，同样很多时候对故障的处理一定要慎之又慎，不能光图快而丢掉一些验证的环节和思考的环节。

那么整体运维如何才能做到高效呢，高效就是让业务操作或业务故障对客户的影响更少，我们需要先从整体的业务操作流程来开始梳理，把每个环节需要做些什么都思考好，能一个人做完的就一个人做，多人协作反而会很低效，主要是沟通和协调的成本就很高，能工具化的就用工具化来支持。整体的自动化操作和标准都是一点一滴积累起来的，之前做自动化时总是想整体来做，做什么框架之类的，其实并不重要，重要的是先把步骤积累和标准积累做好。如果有精力的话可以对过去的步骤做盘点，把能优化掉的尽可能优化掉或取消掉，而且要学会以点带面的思考，尽可能地复制和延续同样的套路和思维。对于标准来说，重要的才添加，不重要的不要急于添加，添加的环节太多未必是好事，对人的依赖度会太高，如果需要添加那么同步的自动化必须要做好，不能因为环节多了效率反而降低了，事故率也变高了。之前发生了事故之后总是先添加环节，后来发现添加的环节多了不但没有治本，反而增加了出现问题的概率，因此需要了解问题背后的原因。

自动化的工作千万不能耽误，能马上做得最好马上就做好，打铁要趁热，做好了自动化再去测试和线上验证，而且做好自动化后要定期回顾和更新。之前做游戏的时候最开始管理的资源有限，业务的操作范围也有限，而且业务架构比较简单，后来随着业务的发展越来越快，业务的类型随之增多而且整体的业务模型也越来越复杂，业务的考核指标越来越高。同样一个工具，架构上或资源上的投入上也需要有相应的变化。原有的自动化方式都是在台中控上操作执行，然后做个多线程的批量化执行就可以了，但是业务量大了后就需要逐台机器自己去执行，中控只需要做汇总就行了。自动化的工具是给人用的，操作起来不能太复杂，人都是懒惰的，能不操作就不操作，能直接一览无余就不要逐个点击，能自动判断的就不要人为判断。其实事故的发生或流程执行不到位很多时候未必就是流程的问题，标准本身就不一定合理，框架也未必对，需要有足够开放的心态，让大家放开了去做，而且整体都不要太急，不要太急于一下子就做成什么样子，自动化是需要时间来积累的。如果我要写一个自动化的工具且不想重新构建，那么找到足够多可靠的小工具就好。自动化未必是工具，而是一种习惯，是需要持续思考的一种方式，要把底层的平台做得足够简单和灵活，组件需要足够的多，然后在这些组件上以搭积木的方式不断地完善平台。假如我们最基本的操作模型就是以中控机 SSH 的方式来连接操作，操作的框架可以是中控执行，或者说是中控发起执行，每遇到一个问题可以通过添加一个脚本或配置就能解决了。如果发现操作环节太多，那

就把中控的脚本编写得交互性强一些，按照交互的方式来分布执行。如果发现执行的方式不够直观，易用性不够好，安全度不够高，那么就做个前端，把前端对接到中控上。如果发现中控不够高效，那就直接对接到业务的服务器上。如果发现操作关联的环节不仅仅局限于服务器，那就把各个业务的接口都开放出来，在此基础上再做一层，支持不同系统之间流程的配置加上各个系统 API 的调用，当然这些都是从操作本身来做优化的，那么我们是否可以从维护的系统和应用本身去做优化呢。之前腾讯推出的 MySQL 就是从底层来做优化把整体版本的更新效率优化掉了。有些时候我们不仅要思考如何实现自动化，还需要思考从哪里来优化会比较合适，从哪里优化会更直接，尝试找到最直接的优化方法，降低操作中存在的风险。可控度和稳定性是基础。就像现在的蓝鲸就做得很好，它们的组件足够可靠，易操作，效率高，随着业务的发展又在组件的基础上做了一层 PAAS。对于实际业务来讲，自动化需要量体裁衣，不能什么都一锅端，适合的才是最好的，可控和稳定永远是前提，逐步积累，如果有能力从底层就做优化将会更佳。

图 13-7 所示的是高效示意图：

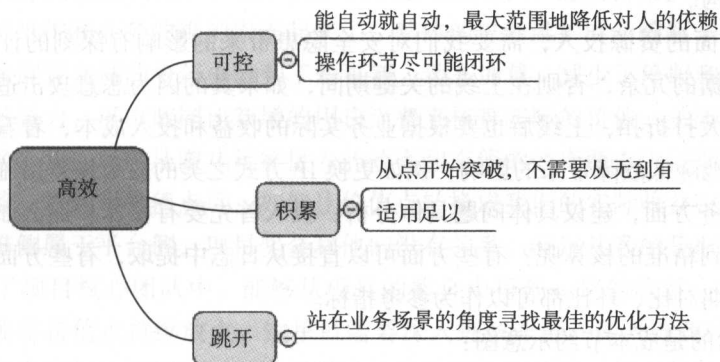


图 13-7 高效示意图

### 13.1.4 成本节约

需要对成本节约的目标有一定的定义，其实成本的核心就是投入产出比，一定要综合考虑，在保障整体游戏安全和稳定的情况下做到整体的投入成本可控。那么如何来控制成本呢，这里需要我们对业务资源的投入有自己的小算盘，对业务的每次故障所带来的影响都能够做到完整的核算。业务资源的投入主要包括计算、存储、网络、安全的资源及运维的增值服务，在计算、存储和网络的资源投入上，前期的评估，资源的投入、配置、冗余管控及回收策略都比较关键。

关于前期的性能选型方面，需要我们先和研发能够先界定好评估的标准，对资源投入的消耗使用率有统一的概念，对客户体验的效果有明确的标准。内部接入产品其实还好，大家相互协作，运营、研发一体化，大家都会对整体成本做控制，标准和对应的评测工具相对都很

容易做出来。而代理产品方面则相对有些困难，这就需要我们尽可能多地积累实际客户案例和效果数据来引导研发做管控。

资源投入方面，对资源投放的力度需要有足够的弹性。例如，关于资源提供的弹性方面，我们之前就和硬件厂商建立了一整套资源的快速支撑方案，这里需要找到尽可能多的资源消耗口，现在有了云就相对简单了很多。业务资源的投放策略是先少量开启，随着业务场景的扩大再逐步来扩充资源的投入。

关于就资源的配置策略方面，前期的配置可以设定得高一些，提供一定的冗余，避免业务架构的死角被忽略，造成硬件性能损耗太大，从而影响业务的正常体验。稳定上线后再根据实际业务高峰期的数据来降低配置，或者尽可能地控制业务架构以实现每个逻辑单元的灵活增减，或者支持将多个逻辑单元整合到一台主机上，对外的端口支持可以灵活配置，数据库也要能够灵活地整合和部署。

关于硬件冗余方面，上线前关键路径至少需要保证双路径的冗余，非核心业务至少能够做到多对 1 的冗余，业务上线后根据实际的故障影响收益和实际的硬件投入成本来做对比，再决策相应的架构。

关于安全方面的资源投入，需要我们对安全隐患带来的影响有深刻的评估，建议前期提前做好临时数据的冗余，否则在上线的关键期间，如果真的因为恶意攻击造成了业务的瘫痪，其效果将会大打折扣，上线后也要根据业务实际的收益和投入成本，看看是否有必要继续投入或尽可能地降低故障带来的影响，例如更换 IP 方式之类的故障恢复措施。

关于增值服务方面，建议具体问题具体分析，投入首先要有核算。那么如何对业务的故障带来的影响做到精准的核算呢？有些方面可以直接从日志中提取，有些方面需要我们发挥对比的价值，同期对比、环比都可以作为参考指标。

图 13-8 所示的是成本节约示意图：

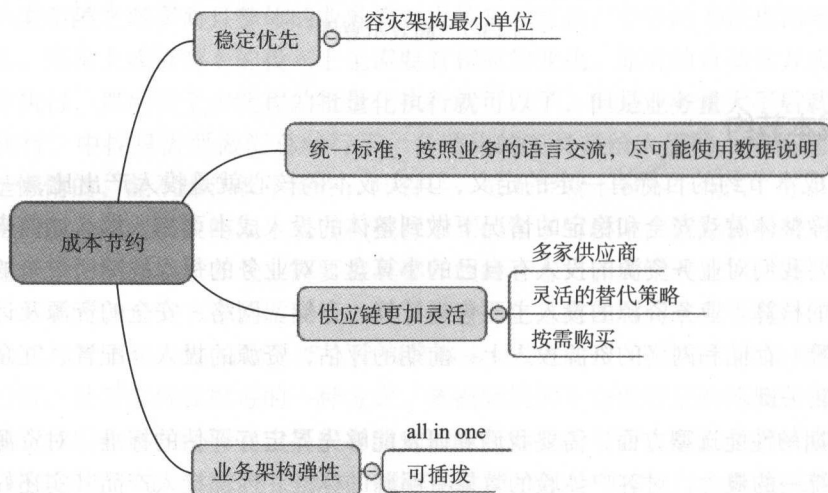


图 13-8 成本节约示意图

## 13.2 游戏运维人的发展

对运维最初的认识就是该行业容易出错,经过一段时间实际的操作之后,发现这个行业简直就是出错的“高危”行业。而且运维往往会处于整个公司价值链的最底层,相对被动,价值难以被量化,有时你会发现干得好是应该的,干得不好就是你的错,比较苦。经过长时间的琢磨,我们做了很多实际的工作来量化运维的价值,例如成本上的节约、效率上的优化与新增收益的转换、体验上的改进与新增收益的转换,等等。成本上的节约很容易理解(人力的节约、硬件的节约都算成本上的节约),关于效率上的优化与新增收益的转换,在这里我可以举个例子,之前我们很关注更新维护对业务带来的影响,如何与能够做到在线更新,如何才能够最大范围地降低更新维护的消耗,如何才能够尽可能地保证每次更新维护的有效性。我们把整体更新维护环节按照内部测试服、线上测试服、线上全量的方式分布更新,保障业务版本的问题能够在内部测试服或线上测试服时就提前发现,经过内部测试服(环境需要绝对的完整),版本发错的问题得到了杜绝,大大降低了版本正式发布的风险(之前曾有过这样的经历,发布到全量失败)。对发布的方式我们做到了一键更新维护,配置文件实现了统一,把数据的维护和业务的维护串连起来了,整体的更新维护时间缩短了50%。关于脚本类的更新和一些配置文件的更新也做到了热更新,动态加载,减少了停服和起服的环节。经过这一番工作的努力,可由此同比新增的用户消费来核算实际的价值。关于体验上的改进与新增收益的转换,我们可以从游戏玩家接入的效率和充值的效率两方面同期对比新增消费来核算实际的价值。找到了价值点后,如何让价值点转换成我们的实际收益呢。我们做了一些工作,之前运维侧属于平台侧,项目奖金和他们没有关系,经过几番的争取,运维核心接口也慢慢纳入到了项目核心团队中,能够从项目的奖金中得到一定的收益。有了价值点的体现,我们就需要将价值点持续放大,这里就需要深入理解业务,深入理解这个行业的深度,并且主动参与各类实际的案例以持续积累经验,主动向上游学习,把自我的价值在公司内部持续放大。

另外一方面,我们也在持续推动如何量化运维人员的行业价值,这里我们借鉴了同行业不同公司对人员能力模型的界定标准及不同公司对运维人员的薪酬标准,慢慢摸索出公司内部关于人员自身行业价值的评价标准,由此也开拓了另外一条价值升值的通道。从目前的行业来看,游戏运维有四个方向:游戏运营侧的技术专家,一方面,能够增加对整体游戏技术的理解,另一方面,能够在游戏的价值输出中提供更多的价值,这里需要他们能够对游戏的测试、评测和研发都有所涉猎,从长远看可以往研发、制作人或运营经理方向发展,离用户最近,价值就放大得越大;游戏运维专家,这里需要他们经历更多产品的游戏运维,对游戏运维梳理总结出一套标准和框架,能够从整体来放大游戏运维的价值;资深游戏运维架构师,转行做售前,将游戏运维的经验和云相结合并输出给客户,为云公司带来收益,由此体现自身的价值;平台运维,将游戏运维的经验下沉和底层的技术相结合,因为他们具备实际的业务经验,对于技术和业务的结合更具优势,而且设计出的技术产品更贴近用户。

图 13-9 所示的为游戏运维人发展示意图：

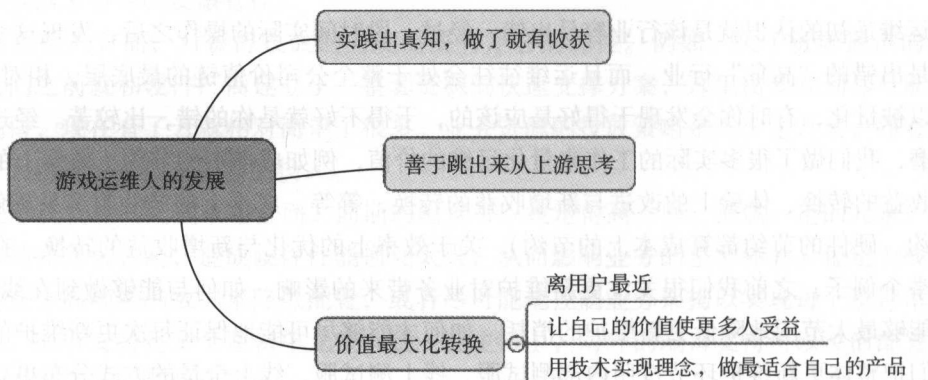


图 13-9 游戏运维人发展示意图

### 13.3 小结

经过几年运维经验的积累和沉淀，游戏运维整体平台智能化的演变从“小米加步枪”转化为蓝鲸运维平台，运维思想从以成本为中心转化为以价值为中心，岗位定位从支撑岗位转化收入岗位，过程中运维人一直在努力寻找可量化自我价值的标准，并主动从自身技能上不断进行突破和改变，尝试了各类方案的验证。随着云时代的到来，游戏运维人和云厂商可以共同为行业转变出谋划策，并逐步摆脱底层建设的依赖，真正实现运维人的价值。



## 数据库平台建设实战

### 作者简介

赵海军，猎豹移动数据库负责人，先后做过 IDC 运维和应用运维，最近三年在做数据库运维，负责公司的 MySQL、RDS、Redis、MongoDB、Memcached 数据库运维，主导设计了公司的 MySQL 数据库平台，并对 800+MySQL 实例进行了迁移，对 LVS 运维、Redis、Codis 运维有一定的经验。博客地址 <http://navyaijm.blog.51cto.com/>。

本章将为大家介绍一下猎豹移动数据库平台相关的一些技术，这个平台也是我亲自参与设计和规划的，很多东西都是从日常工作中总结出来的。在此要感谢我们平台的核心开发工程师李剑辉、邓瑶还有前同事李恒，还要感谢我们数据库团队的刘建，他负责了后期的需求整理和测试。本章不会介绍平台的代码实现，而主要是介绍一下平台的前期规划、架构选型、需求梳理，我觉得任何一个平台若要成功上线，这些都是基础，所以，本章将主要介绍猎豹移动数据库运维的一些规范和这个平台的架构，以及实现了什么样的功能，希望对大家在今后自己的平台化建设方面有所帮助。

### 14.1 规范建立

近年来，平台化、自动化这两个名词在运维的圈子里已经“火”到高潮了，各家公司都在搞平台化、自动化，我们也不例外，但是过程很痛苦，痛苦的根源就是历史包袱太重，对于这点相信很多人都有同感吧。比如软件安装，有使用 yum 安装的，还有编译安装的，并且路径都各不相同，所以，我个人的建议是暂且抛开沉重的历史包袱，建立相关的规范，为平台化建设奠定基础，待我们的平台建设完成之后再逐步瓦解“历史包袱”，逐步向平台上迁



移，实现最终的统一管理。

### 14.1.1 安装规范

说到安装规范，相信维护过大规模数据库的 DBA 都深有感触，我们这边最初也没有规范，安装数据库时有些人喜欢使用 yum 安装，有些人喜欢使用源码编译安装，并且安装目录、数据目录各不相同，这样导致的问题就给后人带来了比较大的运维成本，还不方便进行统一管理，或者给平台化带来了很大麻烦，需要兼容工作情况。

MySQL 安装，我们统一采用“模板”安装，所谓的模板就是将源码用我们指定的编译参数，编译出来一份可以直接启动数据库的文件，需要说明一下的是，我们这边的环境 99% 都是单机多实例部署，大家常见的单机多实例部署都是在服务器上安装好 MySQL，然后用不同的配置文件，启动多个实例，这样做的缺点是多个版本不能共存，而我们通过不同版本的模板部署，多个实例之间不会互相影响（硬件资源除外），单机可以启动任意版本的 MySQL 实例，下面是我们这边 5.5 和 5.6 的编译参数，在此列出以供大家参考。

#### 1. MySQL 5.5 编译参数

Linux 下可用源码编译安装软件，下面给出 MySQL 5.5 的编译参数。

参数详解具体如下。

❑ CMAKE\_INSTALL\_PREFIX:PATH：指定数据库的安装路径。

❑ MYSQL\_DATADIR：指定数据的存储目录。

❑ MYSQL\_UNIX\_ADDR：指定 sock 的文件。

❑ DEFAULT\_CHARSET：默认字符集。

❑ DEFAULT\_COLLATION：默认排序字符集。

❑ WITH\_MYISAM\_STORAGE\_ENGINE：开启 MYISAM 引擎。

❑ WITH\_INNOBASE\_STORAGE\_ENGINE：开启 InnoDB 引擎。

```
cmake -DCMAKE_INSTALL_PREFIX:PATH=/data/mysql/example
-DMYSQL_DATADIR=/data/mysql/example/db
-DMYSQL_UNIX_ADDR=/data/mysql/example/logs/mysql.sock
-DDEFAULT_CHARSET=utf8
-DDEFAULT_COLLATION=utf8_general_ci
-DWITH_MYISAM_STORAGE_ENGINE=1
-DWITH_INNOBASE_STORAGE_ENGINE=1
-DWITH_READLINE=1 -DENABLED_LOCAL_INFILE=1
```

#### 2. MySQL 5.6 编译参数

参数详解具体如下。

❑ CMAKE\_BUILD\_TYPE：指定产品编译说明信息。

❑ CMAKE\_INSTALL\_PREFIX:PATH：指定数据库的安装路径。

❑ MYSQL\_DATADIR：指定数据的存储目录。

- ❑ `DEFAULT_CHARSET`: 默认字符集。
- ❑ `DEFAULT_COLLATION`: 默认排序字符集。
- ❑ `WITH_MYISAM_STORAGE_ENGINE`: 开启 MYISAM 引擎。
- ❑ `WITH_INNOBASE_STORAGE_ENGINE`: 开启 InnoDB 引擎。
- ❑ `WITH_SSL`: 是否支持 SSL。
- ❑ `CMAKE_EXE_LINKER_FLAGS`: 指定使用 `jemalloc` 管理内存。

下面给出 MySQL 5.6 的编译参数:

```
cmake -DCMAKE_BUILD_TYPE=RelWithDebInfo
-DCMAKE_INSTALL_PREFIX=/data/mysql/example
-DMYSQL_DATADIR=/data/mysql/example/db
-DSYSCONFDIR=/data/mysql/example/my.cnf
-DWITH_INNOBASE_STORAGE_ENGINE=1
-DWITH_FEDERATED_STORAGE_ENGINE=1
-DWITH_PARTITION_STORAGE_ENGINE=1
-DDEFAULT_CHARSET=utf8
-DDEFAULT_COLLATION=utf8_general_ci
-DENABLE_DEBUG_SYNC=0
-DENABLED_LOCAL_INFILE=1
-DENABLED_PROFILING=1
-DMYSQL_UNIX_ADDR=/data/mysql/example/logs/mysql.sock
-DWITH_DEBUG=0
-DWITH_SSL=yes
-DWITH_SAFEMALLOC=OFF
-DWITH_BDB=1
-DWITH_blackhole-storage=1
-DCMAKE_EXE_LINKER_FLAGS="-ljemalloc"
```

### 14.1.2 配置规范

关于 MySQL 的配置参数, 我们也可以用统一的模板文件来生成, 当然, 大家会问, 不同配置的机器所用的配置参数值都是一样的吗? 我们这边的环境 99% 都是单机多实例, 所以配置参数值的大小和机器的配置没有必然联系, 所以默认使用的都是我们认为“合适”的值。随着后续库访问量的上升, 我们也会对一些需要调优的参数做调整, 下面给出我们这边所用的 5.5 和 5.6 的 `My.cnf` 参数。

1.`my.cnf` 就是启动 MySQL 的配置参数文件, 默认的配置不太能满足我们线上的需求, 所以有些参数的值需要调整一下, 下面是我们所用的关于 MySQL 5.5 和 MySQL 5.6 的参数说明。

#### 1. `My.cnf` 的重要参数详解

下面来看下 MySQL 5.5 的 `My.cnf` 的参数设置。

- ❑ `key_buffer`: 索引块是缓冲的并且被所有的线程所共享。`key_buffer` 可用于设置索引块的缓冲区大小, 增加它可以得到更好处理的索引 (对所有读和多重写), 可将它增大到你能够负担得起那样多。如果你使它太大, 系统将开始换页并且可能还会变慢。

- ❑ `max_allowed_packet`：服务所能处理的请求包的最大大小及服务所能处理的最大的请求大小，当与大的 BLOB 字段一起工作时该参数相当必要。
- ❑ `sort_buffer_size`：MySQL 执行排序所使用的缓冲大小。如果想要提高 ORDER BY 的速度，首先要看是否可以让 MySQL 使用索引而不是进行额外的排序，如果不能，则可以尝试增加 `sort_buffer_size` 变量的大小。
- ❑ `read_buffer_size`：MySQL 读入缓冲区的大小。对表进行顺序扫描的请求将分配一个读入缓冲区，MySQL 会为它分配一段内存缓冲区。`read_buffer_size` 变量可控制这一缓冲区的大小。如果对表的顺序扫描请求非常频繁，并且你认为频繁扫描进行得太慢，那么可以通过增加该变量值及内存缓冲区的大小来提高其性能。
- ❑ `read_rnd_buffer_size`：MySQL 的随机读缓冲区大小。当按任意顺序读取行时（例如，按照排序的顺序），将分配一个随机读缓冲区。进行排序查询时，MySQL 首先会扫描一遍该缓冲，以避免磁盘搜索，提高查询速度，如果需要排序大量数据，则可适当调高该值。但 MySQL 会为每个客户连接发放该缓冲空间，所以应尽量适当地设置该值，以避免内存开销过大。
- ❑ `thread_cache_size`：这个值（默认为 8）表示可以重新利用保存在缓存中的线程的数量，当断开连接时如果缓存中还有空间，那么客户端的线程将被放到缓存中。如果线程重新被请求，那么请求将从缓存中读取。如果缓存中是空的或是新的请求，那么这个线程将被重新创建。如果有很多新线程，那么增加这个值可以改善系统的性能。通过比较 `Connections` 和 `Threads_created` 状态的变量，可以看到这个变量的作用，根据物理内存设置规则如下：
  - 1G → 8
  - 2G → 16
  - 3G → 32
  - 大于 3G → 64
- ❑ `query_cache_size`：MySQL 的查询缓冲大小（从 4.0.1 开始，MySQL 提供了查询缓冲机制）。使用查询缓冲，MySQL 将 SELECT 语句和查询结果存放在缓冲区中，今后对于同样的 SELECT 语句（区分大小写），将直接从缓冲区中读取查询结果。根据 MySQL 用户手册所述，使用查询缓冲最多可以达到 238% 的效率，通过检查状态值“`Qcache_%`”，可以得知 `query_cache_size` 的设置是否合理——如果 `Qcache_lowmem_prunes` 的值非常大，则表明经常出现缓冲不够的情况；如果 `Qcache_hits` 的值也非常大，则表明查询缓冲使用得非常频繁，此时需要增加缓冲的大小；如果 `Qcache_hits` 的值不大，则表明查询的重复率很低。
- ❑ `log-bin`：打开二进制日志功能，在复制（replication）配置中，作为 MASTER 主服务器必须打开此项，如果需要从最后的备份中做基于时间点的恢复，那么你也同样需要二进制日志。

- ❑ `binlog_format` : 将 binlog 的日志格式设置为 ROW 模式, binlog 有三种日志格式, 分别是 STATEMENT 模式、ROW 模式、MIXED 模式, 具体每种模式的优缺点大家自行参考相关资料, 生产环境中基本上都是 ROW 模式。
- ❑ `log-slave-updates` : 需要做主环境或级联复制的, 需要开启此参数。
- ❑ `wait_timeout` : 服务器关闭非交互连接之前等待活动的秒数。在线程启动之时, 要根据全局 `wait_timeout` 的值或全局 `interactive_timeout` 的值初始化会话 `wait_timeout` 值
- ❑ `interactive_timeout` : 服务器关闭交互式连接之前等待活动的秒数, 交互式客户端定义为在 `mysql_real_connect()` 中使用 `CLIENT_INTERACTIVE` 选项的客户端。默认值为 28 800 秒 (8 小时)
- ❑ `tmp_table_size` : MySQL 的 heap (堆积) 表缓冲大小。所有联合都在一个 DML 指令内完成, 并且大多数联合甚至可以不用临时表即可完成。大多数临时表都是基于内存的 (heap) 表。具有大的记录长度的临时表 (所有列的长度之和) 或包含 BLOB 列的表存储在硬盘上。如果某个内部 heap (堆积) 表的大小超过了 `tmp_table_size`, 那么 MySQL 可以根据需要自动将内存中的 heap 表改为基于硬盘的 MyISAM 表, 还可以通过设置 `tmp_table_size` 选项来增加临时表的大小。也就是说, 如果调高该值, 那么 MySQL 同时将增加 heap 表的大小, 可达到提高连接查询速度的效果。
- ❑ `replicate-wild-ignore-table` : 这个参数是在从库上设置的, 是用来设置需要忽略同步的表, 通过上面的配置文件可以看到, 我们对 MySQL 自带的一些库都设置了忽略同步, 包括 MySQL 库, 也就是说我们的数据库账号、权限信息是不同步的, 这样能最大程度地控制在从库上的误写, 然而如果给从库上加了可写账号那就要另说了。
- ❑ `innodb_file_per_table` : 设置 InnoDB 为独立表空间模式, 每个数据库的每个表都会生成一个数据空间, 独立表空间的优点具体如下。
  - 每个表都有自己独立的表空间。
  - 每个表的数据和索引都会存储在自己的表空间中。
  - 可以实现单表在不同的数据库中移动。
  - 空间可以回收 (除 `drop table` 操作之外, 表空间不能自己回收)。
- ❑ `innodb_buffer_pool_size` : 主要作用是缓存 InnoDB 表的索引、数据、插入数据时的缓冲等信息, 该项设置得越大, 你在存取表中的数据时所需要的磁盘 I/O 就越少, 如果是专用的 MySQL 服务器, 则建议设置为服务器内存的 60% ~ 80%; 如果是单机多实例, 则可视情况而定。
- ❑ `innodb_flush_log_at_trx_commit` : 当 `innodb_flush_log_at_trx_commit` 被设置为 0 时, 日志缓冲将以每秒一次的频率被写到日志文件中, 并且对日志文件做到磁盘操作的刷新, 但是在一个事务中提交不做任何操作; 当这个值为 1 (默认值) 时, 在每个事务提交时, 日志缓冲被写到日志文件, 对日志文件做到磁盘操作的刷新; 当设置为 2 时, 在每个事务提交时, 日志缓冲被写到日志文件, 但不对日志文件做到磁盘操作的刷新。需

要注意的是，并不能保证 100% 每秒都一定会刷新到磁盘，这点还要取决于进程的调度。每次进行事务提交的时候将数据写入事务日志，而这里的写入仅仅是调用了文件系统的写入操作，而文件系统是有缓存的，所以这个写入并不能保证数据已经写入到了物理磁盘。综上所述，设置为 1 最安全，0 性能最好，2 折中，生产环境中建议设置为 2。

❑ `sync_binlog` : `sync_binlog=0` 时，当事务提交后，MySQL 仅仅将 `binlog_cache` 中的数据写入 `binlog` 文件，但不执行 `fsync` 之类的磁盘同步指令，通知文件系统将缓存刷新到磁盘，而让 `Filesystem` 自行决定什么时候来做同步，这个性能是最好的。`sync_binlog=n` 时，在进行 `n` 次事务提交以后，MySQL 将执行一次 `fsync` 之类的磁盘同步指令，通知文件系统将 `binlog` 文件缓存刷新到磁盘；MySQL 中默认的设置是 `sync_binlog=0`，即不做任何强制性的磁盘刷新指令，这时性能是最好的，但风险也是最大的。一旦系统崩溃，文件系统缓存中的所有 `binlog` 信息都将丢失。

❑ `innodb_lock_wait_timeout` : InnoDB 事务在被回滚之前可以等待一个锁定的超时秒数。InnoDB 在它自己的锁定表中自动检测事务死锁并且回滚事务。InnoDB 用 `LOCK TABLES` 语句实现锁定设置。默认值是 50 秒。

下面是我们线上 MySQL 的配置参数，在此列出，仅供参考：

```
[mysqld_safe]
log-error= /data/mysql/example/logs/mysql.log
pid-file= /data/mysql/example/logs/mysql.pid
[client]
port = 3307
socket = /data/mysql/example/logs/mysql.sock
[mysqld]
port = 3307
socket = /data/mysql/example/logs/mysql.sock
key_buffer = 384M
max_allowed_packet = 16M
table_cache = 4096
sort_buffer_size = 2M
read_buffer_size = 2M
read_rnd_buffer_size = 8M
myisam_sort_buffer_size = 64M
thread_cache_size = 8
query_cache_size = 64M
basedir= /data/mysql/example
datadir= /data/mysql/example/db
thread_concurrency = 8
log-bin=mysql-bin
binlog_format = row
log-slave-updates = 1
expire_logs_days = 7
server-id = 103
max_connections=2048
character_set_server=utf8
```

```

wait_timeout=1800
interactive_timeout=1800
skip-show-database
skip-name-resolve
tmp_table_size = 512M
max_heap_table_size = 512M
replicate-wild-ignore-table=mysql.%
replicate-wild-ignore-table=performance_schema.%
replicate-wild-ignore-table=test.%
replicate-wild-ignore-table=information_schema.%
innodb_data_home_dir = /data/mysql/example/db
innodb_file_per_table=1
innodb_log_group_home_dir = /data/mysql/example/db
innodb_buffer_pool_size = 8000M
innodb_additional_mem_pool_size = 20M
innodb_log_file_size = 100M
innodb_log_buffer_size = 8M
innodb_flush_log_at_trx_commit = 2
sync_binlog = 0
innodb_lock_wait_timeout = 80
default-storage-engine = InnoDB
[mysqldump]
quick
max_allowed_packet = 16M
[mysql]
no-auto-rehash
[isamchk]
key_buffer = 256M
sort_buffer_size = 256M
read_buffer = 2M
write_buffer = 2M
[myisamchk]
key_buffer = 256M
sort_buffer_size = 256M
read_buffer = 2M
write_buffer = 2M
[mysqlhotcopy]
interactive-timeout

```

## 2. MySQL 5.6 的 My.cnf

MySQL 5.6 的 My.cnf 的具体代码如下：

```

[client]
port = 10321
socket = /var/mysql/mysql.sock
[mysqld_safe]
socket = /data/mysql/example/logs/mysql.sock
nice = 0
[mysqld]
pid-file = /data/mysql/example/logs/mysql.pid

```



```

basedir                      = /data/mysql/example
bind-address                  = 0.0.0.0
character-set-server          = utf8
character-set-client          = utf8
datadir                      = /data/mysql/example/db
default-storage-engine        = InnoDB
event-scheduler               = ON
expire_logs_days              = 7
general-log                   = 0
log-output                    = FILE
log-error                     = /data/mysql/example/logs/mysqld.log
log-warnings                   = 1
port = 10321
socket                        = /data/mysql/example/logs/mysql.sock
user                          = mysql
server-id = 103
log-slave-updates = 1
skip-name-resolve             = ON
skip-external-locking         = ON
replicate-wild-ignore-table=mysql.%
replicate-wild-ignore-table=performance_schema.%
replicate-wild-ignore-table=test.%
replicate-wild-ignore-table=information_schema.%
long_query_time               = 3
slow_query_log                = ON
log-queries-not-using-indexes = OFF
slow_query_log_file           = /data/mysql/example/logs/slowquery.log
wait_timeout                   = 1800
interactive_timeout            = 1800
connect_timeout                = 5
net_read_timeout              = 120
max_connections                = 2048
max_user_connections          = 2150
key_buffer                    = 64M
max_allowed_packet            = 32M
table_open_cache               = 5000
table_definition_cache        = 500
sort_buffer_size               = 1M
join_buffer_size               = 1M
read_buffer_size               = 1M
read_rnd_buffer_size          = 512KB
thread_cache_size              = 300
tmp_table_size                 = 2G
max_heap_table_size           = 2G
query_cache_type               = 0
innodb-status-file             = 1
innodb_additional_mem_pool_size = 20M
innodb_buffer_pool_size       = 1G
innodb_buffer_pool_instances  = 8
innodb_data_home_dir           = /data/mysql/example/db

```

```

innodb_file_per_table      = 1
innodb_doublewrite         = 1
innodb_flush_log_at_trx_commit = 2
innodb_flush_method        = O_DIRECT
read_buffer_size           = 128K
innodb_lock_wait_timeout   = 120
innodb_log_buffer_size     = 4M
innodb_log_file_size       = 512MB
innodb_log_files_in_group  = 2
innodb_log_group_home_dir  = /data/mysql/example/db
innodb_support_xa          = 0
innodb_thread_concurrency  = 64
innodb_use_sys_malloc      = 0
log-bin                    = /data/mysql/example/db/mysql-bin
max_binlog_size            = 100M
expire_logs_days           = 7
sync_binlog                = 0
binlog_format              = row
relay_log_recovery         = 1
federated
[mysqldump]
quick
quote-names
[isamchk]
key_buffer                 = 16M

```

### 14.1.3 账号、权限规范

我们对账号也做了相关的规范，账号的命名、权限、来源 IP 限制，都有相关的规范约束，为什么要这样做，我大概总结了以下几个要点：

- ❑ 方便运维，通过账号 DBA 就能知道这个账号有什么样的权限。
- ❑ 规范运维，方便后续自动化建设。
- ❑ 降低安全风险，给特定账号对应的权限，监视误操作的发生。

下面用一个具体的例子来说明一下，例如，开发部门申请一个数据库实例，数据库名为 navy\_db，我们为他提供了 3 个账号，一个是用于程序连接的读写账号 navy\_db\_pro，具有增、删、改、查的权限；一个是用于程序连接的只读账号 navy\_db\_sel；还有一个是供开发人员在公司访问时所用的只读账号 navy\_db，每个账号都有严格的来源 IP 限制，不容许有授权网段的账号。

总结起来，有以下几点。

- ❑ 账号命名规范：dbname\_pro 是程序连接的读写账号，dbname\_sel 是程序连接的只读账号，dbname 是办公网连接的只读账号。
- ❑ 授权规范：读账号只有 select 权限，读写账号默认只有 insert、delete、update、select 权限，如果需要其他特殊权限，需要部门领导审批，才能开放。
- ❑ 授权规范：每个账号都对应有严格的来源 IP 限制，禁止授权网段，比如：10.10.10.%。关于数据库授权来源 IP，我了解到好多公司都是授权了整个网段，最开

始我们也是授权整个网段的，但是现在为什么不容许这样做了呢。是这样的，我们曾经有一个项目，最初是 A 团队在做，可能几个月之后又交接给了 B 团队，这都不是重点，重点是他们的工作交接不清楚，还有就是离职交接，关于哪些服务器可以连接这个项目的数据库，对应的账号都有哪些权限，谁都不知道，只能在库里面抓，后来我们就不容许授权整个网段了。如果项目扩容，需要新增服务器，则申请数据库加白名单，这样 DBA 就可以把控整个库的信息了。

#### 14.1.4 目录规范

前面介绍过我们的数据库都是单机多实例部署的，每个实例只有并且只能有一个 database，需要保证目录名称和 dbname 一一对应，那么目录规范的重要性就不言而喻了，图 14-1 是一台服务器上两个数据库实例（navy\_db\_1、navy\_db\_2）的目录结构，14.1.2 一节“配置规范”已经体现了含义：

Basedir	/data/mysql/dbname
Datadir	/data/mysql/dbname/db
innodb_data_home_dir	/data/mysql/dbname/db
innodb_log_group_home_dir	/data/mysql/dbname/db
Socket、log、PID 相关	/data/mysql/dbname/logs

```
[root@属丝运维男 ~]# ll /data/mysql/navy_db_1
total 80
drwxr-xr-x 2 root root 4096 May 14 2012 bin
-rw-r--r-- 1 root root 17987 Jul 14 2011 COPYING
drwxr-xr-x 4 root root 4096 May 14 2012 data
drwxr-xr-x 5 root root 4096 Mar 18 2013 db
drwxr-xr-x 2 root root 4096 May 14 2012 docs
drwxr-xr-x 3 root root 4096 May 14 2012 include
drwxr-xr-x 3 root root 4096 May 14 2012 lib
drwxr-xr-x 2 root root 4096 May 14 2012 logs
drwxr-xr-x 4 root root 4096 May 14 2012 man
-rw-r--r-- 1 root root 1739 Mar 18 2013 my.cnf
drwxr-xr-x 10 root root 4096 May 14 2012 mysql-test
-rw-r--r-- 1 root root 2552 Jul 14 2011 README
drwxr-xr-x 2 root root 4096 May 14 2012 scripts
drwxr-xr-x 27 root root 4096 May 14 2012 share
drwxr-xr-x 4 root root 4096 May 14 2012 sql-bench
drwxr-xr-x 2 root root 4096 May 14 2012 support-files
[root@属丝运维男 ~]# ll /data/mysql/navy_db_2
total 116
drwxr-xr-x 2 root root 4096 May 14 2012 bin
-rw-r--r-- 1 root root 17987 Jul 14 2011 COPYING
drwxr-xr-x 4 root root 4096 May 14 2012 data
drwxr-xr-x 11 mysql mysql 4096 Aug 22 14:05 db
drwxr-xr-x 2 root root 4096 May 14 2012 docs
drwxr-xr-x 3 root root 4096 May 14 2012 include
drwxr-xr-x 3 root root 4096 May 14 2012 lib
drwxr-xr-x 2 mysql mysql 4096 Apr 9 18:40 logs
drwxr-xr-x 4 root root 4096 May 14 2012 man
-rw-r--r-- 1 root root 2041 Oct 20 2014 my.cnf
-rw-r--r-- 1 root root 32963 Mar 19 19:21 mysqloperate.txt
drwxr-xr-x 10 root root 4096 May 14 2012 mysql-test
-rw-r--r-- 1 root root 2552 Jul 14 2011 README
drwxr-xr-x 2 root root 4096 May 14 2012 scripts
drwxr-xr-x 27 root root 4096 May 14 2012 share
drwxr-xr-x 4 root root 4096 May 14 2012 sql-bench
drwxr-xr-x 2 root root 4096 May 14 2012 support-files
[root@属丝运维男 ~]#
```

图 14-1 服务器上 navy\_db\_1、navy\_db\_2 的目录结构

### 14.1.5 其他规范

#### 1. 文件系统及挂载参数

我们推荐文件系统使用 XFS，挂载参数如下：

```
rw,noatime,nodiratime,noikeep,nobarrier,allocsize=8M,attr2,largeio,inode64,swallock
```

关于 XFS 挂载参数的解释，请参考：<http://blog.csdn.net/fangaohua200/article/details/8284019>

#### 2. 本地 MySQL 的操作记录审计

关于 MySQL 的操作记录，下面这个方法适用于你有一个跳板机的情况，你可以从这个跳板机上用 MySQL 客户端登录线上的数据库，那么你执行的所有 MySQL 的命令都会记录下来，保存到一个文件里面。其实现原理就是利用 MySQL 的 prompt 命令把你登录数据库的账号、地址、数据库名还有执行命令的当前时间显示在终端窗口，再配合 tee 把终端窗口的屏显保存到定义的文件里面，然后再去分析这个文件就可以了，具体实现其实很简单，在 My.cnf 里面的 MySQL 区域中添加如下两行参数就能搞定。

具体配置参数如下：

```
[root@运维男 ~]# cat /etc/my.cnf
[mysql]
prompt="\u@\h:\p [\d] : \D >"
tee=/data/log/mysql_history/mysql_history.log
```

#### 3. 文件最大打开数

文件最大打开数，见如下代码：

```
[root@运维男 ~]# cat /etc/security/limits.conf
* soft nproc 10000
* hard nproc 10000
* soft nofile 4194304
* hard nofile 4194304
```

#### 4. 统一内核参数

统一内核参数，见如下代码：

```
[root@运维男 ~]# cat /etc/sysctl.conf
fs.nr_open = 5242880
fs.file-max = 4194304
kernel.core_uses_pid = 1
kernel.msgmax = 1048560
kernel.msgmnb = 1073741824
kernel.shmall = 4294967296
kernel.shmmax = 68719476736
kernel.sysrq = 0
net.core.netdev_max_backlog = 1048576
net.core.rmem_default = 2097152
net.core.rmem_max = 16777216
```

```

net.core.somaxconn = 1048576
net.core.wmem_default = 2097152
net.core.wmem_max = 16777216
net.ipv4.conf.default.accept_source_route = 0
net.ipv4.conf.default.rp_filter = 1
net.ipv4.ip_forward = 0
net.ipv4.ip_local_port_range = 11001      65000
net.ipv4.neigh.default.gc_thresh1 = 10240
net.ipv4.neigh.default.gc_thresh2 = 40960
net.ipv4.neigh.default.gc_thresh3 = 81920
net.nf_conntrack_max = 819200
net.netfilter.nf_conntrack_max = 819200
net.netfilter.nf_conntrack_tcp_timeout_close_wait = 60
net.netfilter.nf_conntrack_tcp_timeout_fin_wait = 120
net.netfilter.nf_conntrack_tcp_timeout_time_wait = 120
net.ipv4.tcp_fin_timeout = 10
net.ipv4.tcp_keepalive_intvl = 15
net.ipv4.tcp_keepalive_probes = 5
net.ipv4.tcp_keepalive_time = 30
net.ipv4.ip_local_port_range = 1024      65000
net.ipv4.tcp_max_orphans = 3276800
net.ipv4.tcp_max_syn_backlog = 1048576
net.ipv4.tcp_max_tw_buckets = 10000
net.ipv4.tcp_mem = 94500000 915000000 927000000
net.ipv4.tcp_orphan_retries = 3
net.ipv4.tcp_reordering = 5
net.ipv4.tcp_retrans_collapse = 0
net.ipv4.tcp_retries2 = 5
net.ipv4.tcp_rmem = 4096      87380    16777216
net.ipv4.tcp_sack = 1
net.ipv4.tcp_synack_retries = 1
net.ipv4.tcp_syncookies = 0
net.ipv4.tcp_syn_retries = 1
net.ipv4.tcp_timestamps = 0
net.ipv4.tcp_tw_recycle = 0
net.ipv4.tcp_tw_reuse = 1
net.ipv4.tcp_window_scaling = 1
net.ipv4.tcp_wmem = 4096      16384    16777216
net.ipv6.conf.all.disable_ipv6 = 1
net.ipv6.conf.default.disable_ipv6 = 1
vm.swappiness = 1

```

对于上述代码中所涉及的重要配置参数，我们来具体解读一下。

- **net.ipv4.ip\_local\_port\_range**：这个参数是预留一段端口范围作为服务（MySQL）的监听端口，默认值是 32 768 ~ 61 000，当你的 MySQL 监听端口大于 1024 的时候，重启 MySQL 服务，你发现经常会有 TCP 连接占用你的 MySQL 监听端口，这让你很恼火，那么设置这个参数吧，把 MySQL 的监听端口放在这个范围里面即可解决上述

问题。

- ❑ `net.ipv4.tcp_max_tw_buckets`: 表示系统同时保持 TIME\_WAIT 套接字的最大数量, 如果超过了这个数字, TIME\_WAIT 套接字将立刻被清除并在 `/var/log/messages` 中打印警告信息, 默认为 180 000, 改为 10 000。
- ❑ `net.ipv4.tcp_tw_recycle`: 开启 TCP 连接中 TIME\_WAIT sockets 的快速回收, 默认是 0, 关闭。
- ❑ `net.ipv4.tcp_tw_reuse`: 设置为 1 表示开启重用, 允许将 TIME\_WAIT sockets 重新用于新的 TCP 连接, 默认是 0, 关闭。
- ❑ `net.ipv4.tcp_fin_timeout`: 表示如果套接字由本端要求关闭, 那么这个参数决定了它保持在 FIN-WAIT-2 状态的时间, 默认值是 60 秒。
- ❑ `net.ipv4.tcp_keepalive_intvl`: 当探测没有确认时, 重新发送探测的频度。默认是 75 秒。
- ❑ `net.ipv4.tcp_keepalive_probes`: 在认定连接失效之前, 发送多少个 TCP 的 keepalive 探测包。默认值是 9。这个值乘以 `tcp_keepalive_intvl` 之后决定了, 一个连接发送了 keepalive 之后可以有多少时间没有回应。
- ❑ `net.ipv4.tcp_keepalive_time`: 当 keepalive 启用的时候, TCP 发送 keepalive 消息的频度。默认是 2 小时。
- ❑ `vm.swappiness`: 默认值是 60, 设置为 0 的时候表示最大限度地使用物理内存, 然后才是 swap 空间, 而不是网上所说的是禁止使用 swap; 设置为 100 的时候表示积极地使用 swap 分区, 并且把内存上的数据及时搬运到 swap 空间里面。

## 14.2 架构设计

本节主要分为两部分, 第一部分给出架构图大概简述一下平台的功能, 第二部分介绍一下平台各个模块的主要功能。

### 14.2.1 架构图

这个平台完成了 MySQL 日常运维 80% 的工作, 包括数据库实例开通、从库扩容、主从切换、权限管理, 等等。

图 14-2 为数据库平台的架构设计。该架构由 LVS、Redir、底层 DB 服务器、管理平台等几个模块组成。LVS 就是一个统一入口的角色, 数据库开通之后, 我们提供给业务人员的地址是 LVS 的虚拟 IP 端口, 业务人员并不了解数据库真正运行在哪台服务器上; Redir 就是提供一个 NAT 功能的工具, 每个库都会有一个公网地址映射到主库, 供业务人员在公司查询数据库; 底层 DB 服务器就是用来运行 MySQL 实例的; 管理平台就是管理调度前面的 3 个模块, 提供 UI 界面供 DBA 操作。



## 14.2.2 各个模块介绍

### 1. LVS

在这个平台中，LVS 充当着入口的角色，即我们对外提供的数据库连接地址是 VIP:Port。一套 LVS 环境（主备、DR 模式）有两个 VIP，一个是读业务的 VIP，一个是写业务的 VIP，然后通过 Port 区分数据库。一个 Port 对应一个数据库，Port 在整个平台中全局唯一，不容许有重复的 Port，平台中可以有多套 LVS 环境。多个从库前面加一层 LVS 转发，做负载均衡调度，这个比较常见，但是对于主库前面加一次 LVS，对大家来说可能有些陌生，下面就来分析一下。关于主库的高可用，最常见的就是 VIP 了，主库挂掉时 VIP 漂移到新的主库，但是在我们这个架构里面，如果用 VIP，那就需要很多 IP（一个库一个 IP）了，不过这都不是问题，其实最主要的问题是维护成本大。IP 变动还得更新运维的资产系统，如果忘记更新了，IP 一旦被其他机器抢占，就会出大问题。还有就是我们的这个方案，主库没有做真正的高可用，当主库挂掉时，需要在平台上点一下迁移按钮，才会做故障迁移，如果直接把主库的地址提供给业务人员使用，那么发生迁移之后主库的地址变了，还需要通知业务改代码，更换主库的地址，这显然是不科学的，这也是我们在主库迁移前加一层 LVS 的一个主要原因。

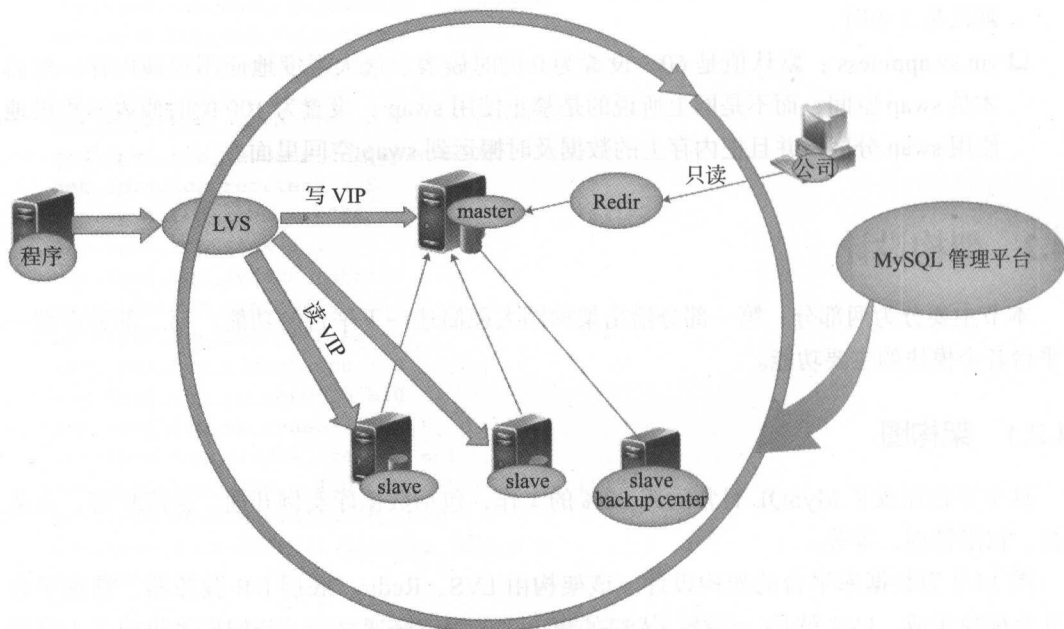


图 14-2 数据库平台架构图

### 2. Redir

Redir 是一个特别轻量级的端口映射工具，从图 14-2 中可以看到，我们这边需要从公司内网访问数据库，而我们的数据库服务器都是内网地址，所以就需要一个端口映射工具了，

然后结合小米开源的进程管理工具 god，实现端口映射的进程监控和维护，当然也可以用 iptables 实现端口映射功能，或者其他的开源工具。

### 3. 底层 DB 服务器

底层的 DB 服务器就是整个平台的资源池，用于部署 MySQL 实例。DB 服务器分为两类，一类是提供线上的 MySQL 实例部署；另一类是提供备份的 MySQL 实例部署。也就是说只要在我们的平台上开通一个库，至少要有一主库、两从库，一个从库用于候选主库，另一个用于备份，备份 DB 服务器上的 MySQL 从库不提供服务，只是用作备份。我们的备份策略都是在备份专用的从库上进行的，冷备 + 增量备份。

### 4. 管理平台

管理平台类似于一个调度中心，管理调度 LVS、Redir、DB 服务器资源，底层使用的 SaltStack，管理平台是 master 角色，其他都是 client 角色；那么实现这个平台的语言当然是采用大家熟悉的 Python 了，Web 框架也是大家常用的 Django。

## 14.3 功能介绍与实践

### 14.3.1 操作部分

这部分是整个平台的核心功能，日常运维的一些功能都包含在这里，包括新增实例、从库、迁入（把一个库从平台外迁入平台管理）、实例下线、迁移、权限管理等，下面将逐一介绍各个功能。

#### 1. 新增实例、从库

如图 14-3 所示的是新增一个数据库的操作截图，接下来逐一说一下各个选项的含义。

**业务名称**，这个就是数据库的名字，在 14.1.4 节的“目录规范”中已经介绍过，每一个实例只能有一个 database，并且 database 的名字和目录的名字一一对应，与平台对应的业务名称也有一定的限制，比如，整个平台的业务名称（数据库名）是唯一的，还有就是数据库名不支持的特殊符号，我们在平台上也是做了限制的。

**数据库版本**，这个是可以自己选择的，前提是需要我们在后台把相应的版本模板部署好，然后在平台上录入版本号，接下来新建实例的时候就可以选择你的新版本了，个人觉得这个功能很实用，只要你把模板部署好，无论是 Percona 还是 MariaDB，还是官方的都可以安装。

**Master 机器 IP**，就是这组实例的主库所在的 DB 服务器 IP，需要自己选择，底层的 DB 服务器有几十台甚至上百台，每台的配置也不尽相同，所以选择一台合适的 DB 服务器也是一个体力活，我们正在考虑优化，比如平台可以动态推荐给我们低负载的 Top5 供我们选择，而不是在茫茫 IP 列表里去选择合适的 DB 服务器。

业务名称: navy\_db bbs数据库

数据库版本: 5.6.19

机房: 兆维

Master机器IP: 10.10. [redacted] 如需重新定义, 请重选

Master机器端口: 10749

LVS宿主机: 10.10. [redacted] 如需重新定义, 请重选

LVS映射端口: 10749 LVS映射过来的端口

redir映射端口: 10749 redir映射过来的端口

Slave机器IP: 10.10. [redacted] 如需重新定义, 请重选

是否开启主从分离: ☐

冷备机器IP: 10.10. [redacted] 如需重新定义, 请重选

用途: navy测试数据库

使用级别: 中等

业务使用人: 赵海军

使用人手机: 13813813800

使用人邮箱: navy@cmcm.com

提交

图 14-3 新增数据库

端口, 这个是平台全局把控的, 并且整个平台全局唯一, 所有端口都是平台自己生成的, 而且不可以修改, 包括 Master 机器端口、LVS 映射端口、Redir 映射端口。除此之外, 我们还规定, 一个实例用一个端口, 相信大家从图 14-3 中已经看出来了。

**LVS 宿主机**, 在上面功能模块介绍的小节中, 已经介绍过了 LVS 的内容, 我们本着不把所有鸡蛋放在一个篮子中的原则, 一个平台有多套 LVS 环境, 这个地方就是让你选择你这组实例要选用哪组 LVS, 这里选择的 IP 是 LVS 调度 Master 的 IP。

**Slave 机器 IP**, 就是选择从库所在的 DB 服务器, 和 Master 机器 IP 的功能一样。

**是否开启主从分离**, 如果这个库的读压力很大, 业务方要求开通读写分离, 那么需要把这个勾选上, 部署完成之后, LVS 的读 VIP 就会转发到从库 (默认不开启读写分离的话, LVS 的写 VIP 和读 VIP 都会转发到主库)。另外, 读写分离功能需要程序支持, 我们会为业务方提供两个地址, 一个写地址, 一个读地址, 估计很多人又要疑惑了, 市面上有很多开源的 DBproxy 都支持读写分离, 为什么你们不用呢。首先, 我个人觉得在程序上实现读写分离是最靠谱的方案了, 没有之一, 前提是你得想办法让开发改代码支持读写分离; 其次, 我们也在测试调研各种开源的 DBproxy, 后期可能会让平台自身支持读写分离。

**冷备机器 IP**，就是上面介绍的 DB 服务器的另一类备份服务器，冷备份服务器上的从库不提供业务，只是用来备份，想必大家都会说：“哇，你们好土豪啊，还有专门的备份从库”，好吧，我只能哈哈了。

**新增从库**，如图 14-4，线上 zabbix 这个库现在压力比较大，主库已经不能承受住所有的读写请求，现在需要扩展一个从库，实例名，选择 zabbix 这个库；新从库的 IP，选择这个从库需要放在哪个 DB 服务器上。

图 14-4 扩容从库

关于新添加的从库是否加入 LVS 只读列表中这个选项，我重点说明一下。

首先，我们加从库的原理是把备份服务器的从库停掉，将文件拉到你所选择的新从库的 IP 服务器上启动。

然后在主库上加一个主从同步账号，授权给新从库的 IP，这时，新的从库会自动连主库（主从同步状态，两个 yes），然后启动备份服务器上的实例，如果新的从库不能启动或主从同步状态不好，那么平台返回添加从库失败，反之，平台返回添加从库成功。



注意

这里需要注意的是平台不会去判断新加的从库是否有延时，并且新加的从库也不会有账号信息，所以，如果之前没有开启过读写分离的库，现在要加一个从库开启读写分离的话，我们就可以勾选此选项，待从库的账号添加完成，确认新加的从库没有延时，最后把 LVS 读的地址提供给业务方；如果线上之前已经有一个库开启了读写分离，现在读的压力大，那就需要扩展一个从库，这样的场景下这个选项一定不能勾选。

## 2. 迁入

迁入，就是把平台之外的库迁入到平台管理中，这个平台一上线，我们就经历了一次大迁移，把将近 500 多个实例迁移到了这个平台上，历时半年之久，如图 14-5 所示，其他选项和新增实例一样，这里就不再阐述了，下面重点说一下备份地址这个选项。

从图 14-5 中不难看出，这是一个 rsync 的路径，1.1.1.10 是 rsync 服务端机器的 IP，MySQL 是 rsync 的模块名称，navy\_db\_2 是我们要迁移的这个库的目录，navy\_db\_2 是一份

可用的从库的文件，什么是可用的从库文件，关于这点估计很多人又会有疑惑，下面我用具体的例子来说明一下吧。

备份地址: 1.1.1.1:mysql\_bak/navy\_db 事先手动同步好的rsync地址

业务名称: navy\_db 例如: bbs

数据库版本: 5.6.19

机房: [dropdown]

Master机器IP: 10.10 [dropdown] 如需重新定义, 请重选

Master机器端口: 10750

LVS宿主机: 10.10 [dropdown] 如需重新定义, 请重选

LVS映射端口: 10750 LVS映射过来的端口

redir映射端口: 10750 redir映射过来的端口

Slave机器IP: 10.10 [dropdown] 如需重新定义, 请重选

是否开启主从分离: ☒

冷备机器IP: 10.10 [dropdown] 如需重新定义, 请重选

用途: navy测试数据库

使用级别: 中等

业务使用人: 赵海军

使用人手机: 1381380000

使用人邮箱: navy@cmcm.com

执行!

图 14-5 迁入

例如，需要将 navy\_db\_2 这个库迁移到平台上，那么现在需要对这个库按照我们相关的安装配置规范做一个从库，放在 1.1.1.10 上，然后给 1.1.1.10 上配置 rsync 服务端，模块名是 MySQL，路径指定为 /data/mysql/，然后再把这个从库停掉，那么 /data/mysql/navy\_db\_2 就是一份可用的从库文件，然后就可以在平台上迁入了，迁入完成后，需要在 navy\_db\_2 这个主库上添加一个主从同步的账号授权给平台上的主库，现在就是 A（平台外的主库）← B（平台上的主库）← C（平台上的从库）这样的同步关系了，待程序把程序的读写切换到平台提供的地址上时，B 和 A 的同步关系断开，就完成了 navy\_db\_2 这个库的迁移。程序从老地址切换到新地址，中间有很多细节问题，在这里就不做过多的说明了。

### 3. 调整

如图 14-6，调整这个模块包括从库的下线迁移、主库的下线迁移和冷备库的迁移这些内容，从图 14-6 可知，冷备库只能迁移和不能下线内容，是的，我们规定过，只要是线上的实例必须要有冷备库，因为备份操作就是在冷备库上进行的。



10记录/页

过滤zabbix

应用	主机IP	端口	角色	server-id	删除	迁移	强制迁移
zabbix_proxy	10.57.65.37	10482	冷备库	103		迁移	
zabbix_proxy	10.57.65.27	10482	从库	102	删除	迁移	
zabbix_proxy	10.57.65.26	10482	主库	101	删除	迁移	迁移

图 14-6 DB 实例调整

(1) 下线从库

下线从库，如图 14-6 所示，选择你需要下线的从库，点击删除即可，然后平台会把这个从库停掉，并且在目录上加上当前时间戳重命名，而非直接删除。

(2) 迁移从库

如图 14-7 所示，大概的迁移流程步骤如下。

- 步骤 1：选择待迁移的从库。
- 步骤 2：选择新的从库所在的 DB 服务器 IP。
- 步骤 3：停冷备库实例。
- 步骤 4：推送冷备库实例文件到新的从库服务器。
- 步骤 5：启动冷备库。
- 步骤 6：启动新的从库。
- 步骤 7：主库上授权主从同步账号给新的从库 IP。
- 步骤 8：判断主从一致性。
- 步骤 9：判断老的从库是否在 LVS 读的 VIP 下面，如果在则剔除。

迁移从库

业务名称 zabbix

待迁移库IP 10.10.10.10

新的从库IP 10.10.10.10

执行！

图 14-7 迁移从库

- 步骤 10：关闭老的从库。
- 步骤 11：在主库上取消对老库的主从同步授权。
- 步骤 12：迁移完成。

(3) 迁移主库

迁移主库即计划内的迁移，比如主库所在的 DB 服务器需要关机维护，这个时候就需要做计划内的迁移了，当然，我们这种迁移方式是需要业务中断的，不过时间很短，分钟级别的，大概流程如下。

- 步骤 1：选择待迁移的主库点击迁移跳至图 14-8。
- 步骤 2：选择新主库的 DB 服务器 IP。
- 步骤 3：判断所有从库和主库的主从状态和一致性。
- 步骤 4：停止备份服务器的实例。
- 步骤 5：推送冷备服务器的数据到新主库的 IP。

迁移主库

业务名称 zabbix

待迁移库IP 10.10.10.10

新主库IP 10.10.10.10

执行！

图 14-8 迁移主库



步骤 6：启动备份服务器的实例。

步骤 7：启动新主库的实例。

步骤 8：老的主库添加主从同步账号，授权给新的主库。

步骤 9：判断主从同步数据一致性。

步骤 10：把老主库的账号权限添加到新的主库。

步骤 11：在新主库上添加老的从库的主从同步账号。

步骤 12：删除 LVS、Redir 到老主库的映射。

步骤 13：判断主从同步数据一致性。

步骤 14：停止新的从库，删除 master.info 文件。

步骤 15：启动新的主库。

步骤 16：把老的从库 change master to 到新的主库。

步骤 17：判断主从同步数据一致性。

步骤 18：添加 LVS、Redir 到老主库的映射。

步骤 19：停掉老的主库并重命名命令。

步骤 20：迁移完成。

#### (4) 强制迁移主库

强制迁移主库即计划外的迁移，比如一个库突然挂掉了或底层的 DB 服务器硬件出问题了，短时间内无法恢复。我们线上真出现过服务器主板挂掉的情况，当时就用到了强制迁移主库的功能，如图 14-9 所示，大概迁移流程如下。

在说流程之前首先再次强调一下备份从库的概念，备份从库就是运行在冷备服务器上的从库，不提供业务，只是用于备份，所有的备份操作都是在这个从库上进行的。

步骤 1：判断从库和备份从库同步主库的位置点哪个最新（Relay\_Master\_Log\_File、Exec\_Master\_Log\_Pos），结果有 3 种情况，备份从库和普通从库同步主库的点是一样的、备份从库同步主库的点最新、其中一个普通从库同步主库的点最新。

步骤 2：如果步骤 1 的结果是备份从库和普通从库同步主库的点是一样的，那么随机找一个从库提升为主库，对备份从库和其他普通从库执行如下的命令重新指向新的主库。

```
change master to
master_host='master_ip',
master_user='slave',
master_password='123456',
master_log_file='主库 binlog 的位置',
master_log_pos=主库 binlog 的 pos 点;
```

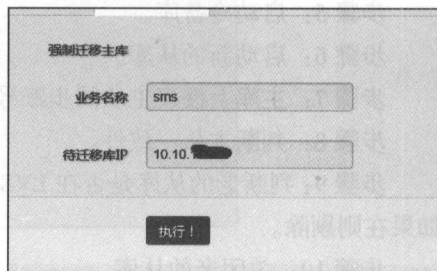


图 14-9 强制迁移主库

步骤 3：把故障主库的账号权限添加的新的主库上。

步骤 4：把故障主库的 LVS、Redir 映射删掉，并映射到新的主库上。

步骤 5：如果步骤 1 的结果是备份从库同步主库的点最新，那么把其他所有的普通从库都停掉并且把目录重命名带上时间戳，用备份从库的数据重新建立一组主从环境，随机选择一个旧从库的地址作为新的主库。

步骤 6：把故障主库的 LVS、Redir 映射删掉，并映射到新的主库上。

步骤 7：如果步骤 1 的结果是其中一个普通从库同步主库的点最新，那么就这份从库的数据为基础，重新做备份从库。

步骤 8：把这个从库提升为主库。

步骤 9：把故障主库的 LVS、Redir 映射删掉，并映射到新的主库。

步骤 10：迁移完成。

#### (5) 迁移冷备库

这个功能就是迁移备份从库，比如备份 DB 服务器磁盘或 IO 快到瓶颈了，我们需要把一部分库迁移到其他备份 DB 服务器上时，就使用这个功能。这个功能也很简单，就是把当前的数据库实例停止推送到新的备份 DB 服务器上启动，然后把老的目录加上时间戳重命名就完成了迁移，如图 14-10 所示。

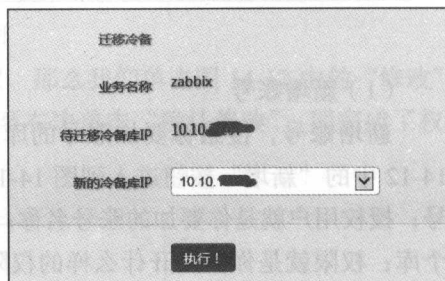


图 14-10 迁移冷备库

#### (6) 下线主库

下线主库会把从库、冷备库还有其他对应的资源都做下线处理，当然，这个下线只是下线而已，不做删除操作，如图 14-11 所示的，是操作日志，我们可以看到一些具体的信息，这些信息都是入库的，方便后续的资源回收分析及审计使用。

业务名称 navy56				
任务名称 删除一组实例				
任务描述 删除一组实例，主库：10.10.10.10 从库：[u'10.10.10.10', u'10.10.10.10']				
时间	主机	任务	状态	描述
2015-08-26 19:43:28	10.10.10.10	关闭数据库实例	OK	
2015-08-26 19:43:30	10.10.10.10	移除数据库实例	OK	backup_name navy56.20150826194328
2015-08-26 19:43:30	10.10.10.10	取消数据库主从授权	OK	
2015-08-26 19:43:35	10.10.10.10	关闭数据库实例	OK	
2015-08-26 19:43:36	10.10.10.10	移除数据库实例	OK	backup_name navy56.20150826194335
2015-08-26 19:43:36	10.10.10.10	取消数据库主从授权	OK	
2015-08-26 19:43:45	10.10.10.10	关闭数据库实例	OK	
2015-08-26 19:43:46	10.10.10.10	移除数据库实例	OK	backup_name navy56.20150826194345
2015-08-26 19:43:46	10.10.10.10	删除redir映射	OK	
2015-08-26 19:43:46	10.10.10.10	删除一组数据库实例完成	JOB_OK	

图 14-11 下线主库

4. 权限管理

这部分的主库功能包括新加账号、删除账号、修改密码、修改权限等，有了这个功能再也不用登录 MySQL 操作了，是不是很方便呢，下面就为大家依次曝光功能点，如图 14-12 所示：

zabbix

搜索：实例名

GO

IP	端口	实例名	实例角色	授权用户	授权ip	权限	权限数据库	权限表	操作
10.10.19.97	10392	zabbix	主库	slave	10.10.19.96	REPLICATION SLAVE	*	*	<div>修改</div> <div>删除</div>
10.10.19.97	10392	zabbix	主库	slave	10.10.1.197	REPLICATION SLAVE	*	*	<div>修改</div> <div>删除</div>
10.10.19.97	10392	zabbix	主库	zabbix_pro	10.10.15.28	ALL	zabbix	*	<div>修改</div> <div>删除</div>
10.10.19.96	10392	zabbix	从库	zabbix_sel	10.10.15.28	SELECT	zabbix	*	<div>修改</div> <div>删除</div>

新增

修改密码

图 14-12 权限管理

(1) 新增账号

新增账号，搜索你要加账号的库，比如我们需要给 zabbix 这个库加一个账号，单击图 14-12 中的“新增”按钮进入到图 14-13，主机地址就是你需要给哪个 DB 服务器的库添加账号；授权用户就是你要加的账号名称；授权地址就是你授权哪个 IP 可以通过这个账号访问这个库；权限就是你需要开什么样的权限给这个账号，这里我们把 MySQL 常用的权限都放在列表里面，你只需要把你需要开放的权限从左边拖到右边即可；数据库就是你授权这个账号访问哪个库；表就是你授权哪个表给这个账号。

新增授权：

业务名称：zabbix

主机地址：10.10.19.97

授权用户：navy\_sel

授权地址：8.8.8.8 多个ip请用“,”隔开

权限：

INSERT  
SHOW VIEW  
SUPER  
UPDATE

SELECT

数据库：zabbix

表：\*

取消

确认增加

图 14-13 新增授权

(2) 修改已有账号密码

比如我们想修改刚才新的账号 navy\_sel 的密码，那么我们单击图 14-12 中的“修改：密

码按钮跳转至图 14-14，输入授权账号 navy\_sel，输入授权地址 8.8.8.8，输入新的密码，单击“确认修改”，即完成了密码修改。

**修改用户密码：**

业务名称：zabbix

授权用户：navy\_sel

授权地址：8.8.8.8

新密码：\*\*\*\*\*

取消 确认修改

图 14-14 修改密码

### (3) 修改权限

比如刚才开通的 navy\_sel 这个账号需要 create 权限，那么我们单击图 14-12 中的“修改”按钮跳转至图 14-15，从权限列表里面把 create 权限拖至右边单击“确认修改”，即完成了权限修改，立即生效。

**修改授权属性：**

业务名称：zabbix

主机地址：10.10.

授权用户：navy\_sel

授权地址：8.8.8.8

权限：ALTER, INSERT, CREATE ROUTINE, DROP

SELECT, CREATE

数据库：zabbix

表：\*

关闭 确认修改

图 14-15 修改权限

## 14.3.2 日志部分

本节主要是为大家介绍一下这个平台与日志相关的功能，包括 3 个模块：平台操作日志、数据库账号管理日志和 mysqld.log 日志，当然还有慢查询日志，不过，慢查询日志分析是另外一个平台，后期会集成在这个平台上，后面的 14-4 会讲到。

1. 平台操作日志

平台操作日志这个日志模块会记录新建数据库、删除数据库、修改 LVS 映射、下线数据库、迁移数据库、平台上新增账号、删除账号等一系列操作相关的日志，如图 14-16 所示：

100	记录/页	过滤: navy					
操作时间	业务名称	操作者	任务名称	任务状态	任务描述	任务详情	
2015-08-26 19:43:20	navy55	zhaohajun	删除一组实例	成功	删除一组实例, 主库: 10.10.1. 从库: [u'10.10.1.', u'10.10.1.'].	详情	
2015-08-26 19:42:34	navy561	zhaohajun	删除一组实例	成功	删除一组实例, 主库: 10.10.1. 从库: [u'10.10.1.', u'10.10.1.'].	详情	
2015-08-26 19:31:41	navy561	zhaohajun	创建一组实例	成功	主库: 10.10.1. 从库: 10.10.1. 冷备库: 10.10.1. 端口: 10421.	详情	

图 14-16 平台日志

单击图 14-16 中的“详情”还可以看到具体操作的内容，如图 14-17 所示：

业务名称 navy56				
任务名称 创建一组实例				
任务描述 主库: 10.10.1. 从库: 10.10.1. 冷备库: 10.10.1. 端口: 10415				
时间	主机	任务	状态	描述
2015-08-26 19:05:15	10.10.1.	安装前环境监测	OK	
2015-08-26 19:05:16	10.10.1.	安装前环境监测	OK	
2015-08-26 19:05:18	10.10.1.	安装前环境监测	OK	
2015-08-26 19:05:20	10.10.1.	安装前环境监测	OK	
2015-08-26 19:05:20	10.10.1.	开始安装数据库实例	OK	
2015-08-26 19:05:43	10.10.1.	安装数据库实例	OK	
2015-08-26 19:05:53	10.10.1.	启动数据库实例	OK	
2015-08-26 19:05:54	10.10.1.	安装前环境监测	OK	
2015-08-26 19:05:54	10.10.1.	开始安装数据库实例	OK	
2015-08-26 19:06:14	10.10.1.	安装数据库实例	OK	
2015-08-26 19:06:29	10.10.1.	启动数据库实例	OK	
2015-08-26 19:06:33	10.10.1.	安装前环境监测	OK	
2015-08-26 19:06:33	10.10.1.	开始安装冷备库实例	OK	
2015-08-26 19:06:54	10.10.1.	安装冷备库实例	OK	
2015-08-26 19:07:04	10.10.1.	启动数据库实例	OK	
2015-08-26 19:07:04	10.10.1.	创建用户	OK	
2015-08-26 19:07:04	10.10.1.	数据库主从授权	OK	
2015-08-26 19:07:05	10.10.1.	创建用户	OK	
2015-08-26 19:07:05	10.10.1.	数据库主从授权	OK	
2015-08-26 19:07:05	10.10.1.	SHOW MASTER STATUS	OK	
2015-08-26 19:07:05	10.10.1.	关闭SLAVE	OK	
2015-08-26 19:07:05	10.10.1.	建立数据库主从	OK	
2015-08-26 19:07:05	10.10.1.	启动SLAVE	OK	
2015-08-26 19:07:05	10.10.1.	关闭SLAVE	OK	
2015-08-26 19:07:05	10.10.1.	建立数据库主从	OK	
2015-08-26 19:07:05	10.10.1.	启动SLAVE	OK	
2015-08-26 19:07:12	10.10.1.	添加redis映射	OK	
2015-08-26 19:07:12		安装一组数据库实例完成	JOB_OK	

图 14-17 新建实例日志

2. 数据库账号管理日志

数据库账号管理这个日志模块会记录通过平台向数据库中添加账号、删除账号的详细信息，如图 14-18 所示：



10 记录/页

过滤 zabbix

编辑时间	操作用户	操作	实例名	实例IP	授权用户@授权HOST	授权库.授权表	权限
2015-09-02 11:10:07		添加权限	zabbix	10.10.10.10	zabbix_sel@10.10.10.10	zabbix.*	SELECT
2015-09-02 10:14:36		删除权限	zabbix	10.10.10.10	navy_sel@8.8.8.8	zabbix.*	SELECT
2015-08-31 10:07:38		删除权限	zabbix	10.10.10.10	admin@10.10.10.10	zabbix.*	ALL
2015-08-31 09:59:10		添加权限	zabbix	10.10.10.10	admin@10.10.10.10	zabbix.*	ALL
2015-08-29 16:48:41		添加权限	zabbix	10.10.10.10	navy_sel@8.8.8.8	zabbix.*	激活 SELECT 转到“电脑设置”以激活 Windows。

图 14-18 权限管理日志

### 3. mysql.log 日志

mysql.log 这个日志模块会展示线上所有数据库实例最近 10 分钟的 mysql.log，后台是每 10 分钟切割一次线上数据库实例的 mysql.log 并把数据归档供前端展示，有了这个功能后再也不用登录服务器查看 MySQL 的错误日志了，如图 14-19 所示：

50 记录/页

过滤: zabbix

实例名	主机IP	文件名字	创建时间	详细LOGS
zabbix	10.10.10.10	10.10.10.10 zabbix.txt	2015-09-04 10:02:42	LOGS
zabbix	10.10.10.10	10.10.10.10 zabbix.txt	2015-09-04 10:00:02	LOGS
zabbix	10.10.10.10	10.10.10.10 zabbix.txt	2015-09-04 10:00:01	LOGS
zabbix	10.10.10.10	10.10.10.10 zabbix.txt	2015-09-04 09:52:42	LOGS
zabbix	10.10.10.10	10.10.10.10 zabbix.txt	2015-09-04 09:50:01	LOGS
zabbix	10.10.10.10	10.10.10.10 zabbix.txt	2015-09-04 09:50:01	LOGS
zabbix	10.10.10.10	10.10.10.10 zabbix.txt	2015-09-04 09:42:42	LOGS
zabbix	10.10.10.10	10.10.10.10 zabbix.txt	2015-09-04 09:40:02	LOGS

显示第 1 至 8 项记录，共 8 项 (由 1,500 项记录过滤)

上页下页

图 14-19 mysql 错误日志

选中你想要查看实例的“详细 LOGS”查看内容，如图 14-20 所示：

#### 14.3.3 资产部分

资产主要分为两个部分，第一部分是实例信息的一些基本信息，比如，库的负责人、负责人的联系方式、用途、版本、IDC 等；第二部分是平台里面所有的 DB 服务器列表。



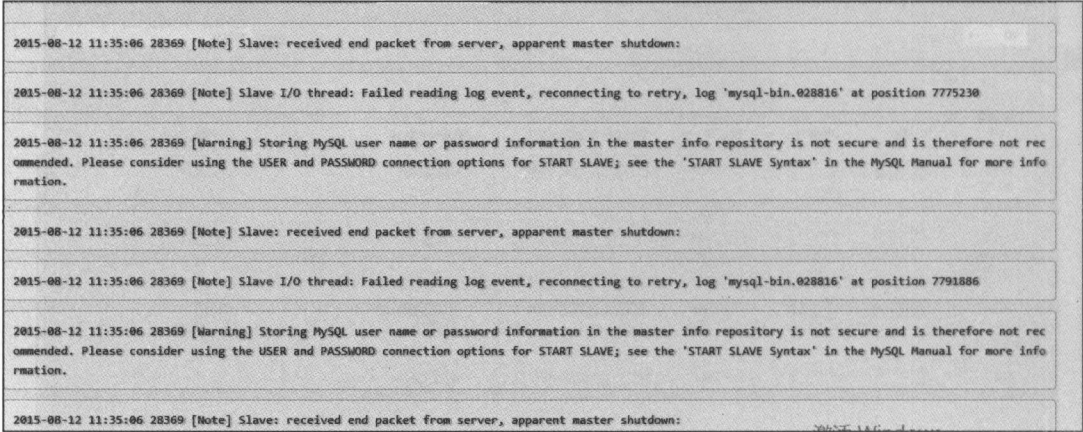


图 14-20 mysqld 错误日志详细内容

1. 实例列表

如图 14-21 所示的就是实例信息的展示，支持模糊搜索，你可以根据实例名、负责人、IDC 等一些信息进行搜索，还可对一些基础信息做变更，比如有一个库的用途和负责人都发生了变更，那么你可以单击“编辑”做相应的变更。

10 记录/页 过滤

实例名	用途	版本	IDC	使用人	使用级别	是否邮件	是否短信	邮件地址	短信地址	编辑
zabbix	zabbix master	5.6			重要	是	是			编辑
wxfv_dcs	None	5.5				是	是			编辑
wx_dkf	None	None				是	是			编辑
wprm	None	5.6				是	是			编辑

激活 Windows

图 14-21 实例列表

2. 服务器列表

如图 14-22 所示的就是服务器列表的展示，这里只展示 DB 服务器，也就是运行 MySQL 实例的服务器，包括冷备服务器，关于“用途”这一列解释一下，就是记录一下这个服务器运行的具体是线上 MySQL 实例还是备份 MySQL 实例，还有一些服务器是 PCIE-SSD 卡的，都记录在“用途”这一列里。

主机IP	用途	IDC	创建时间
10.10.10.27	数据库备份节点	WX	五月 29, 2015 3:52 p.m.
10.10.10.29	数据库节点(主用)	WX	十二月 16, 2015 9:56 a.m.
10.10.10.27	数据库节点(主用)	WX	五月 29, 2015 3:51 p.m.
10.10.10.28	数据库节点(主用)	WX	十月 20, 2015 4:20 p.m.
10.10.10.24	数据库节点(主用)	WX	一月 4, 2016 10:23 a.m.
10.10.10.18	数据库节点(主用)	WX	五月 29, 2015 3:49 p.m.
10.10.10.10	数据库节点	LF	三月 15, 2016 11:09 p.m.
10.10.10.27	数据库节点	LF	三月 15, 2016 11:09 p.m.
10.10.10.12	数据库节点	LF	三月 15, 2016 11:09 p.m.
10.10.10.15	None	ZW	四月 2, 2015 10:30 p.m.

图 14-22 服务器列表

### 14.3.4 信息展示

信息展示这个模块主要是以一个全局的视野展现整个平台的相关信息，包括 3 个部分：实例信息关系、主机实例信息、数据权限信息。这个模块也是我们 DBA 日常工作经常会用到的，比如查看一下某个数据库有哪些账号，哪些服务器可以连到这个数据库，它的 LVS 映射地址是多少；主机实例信息可以查看一个服务器上有多少个 MySQL 实例、多少主库、多少从库等。

#### 1. 实例信息关系

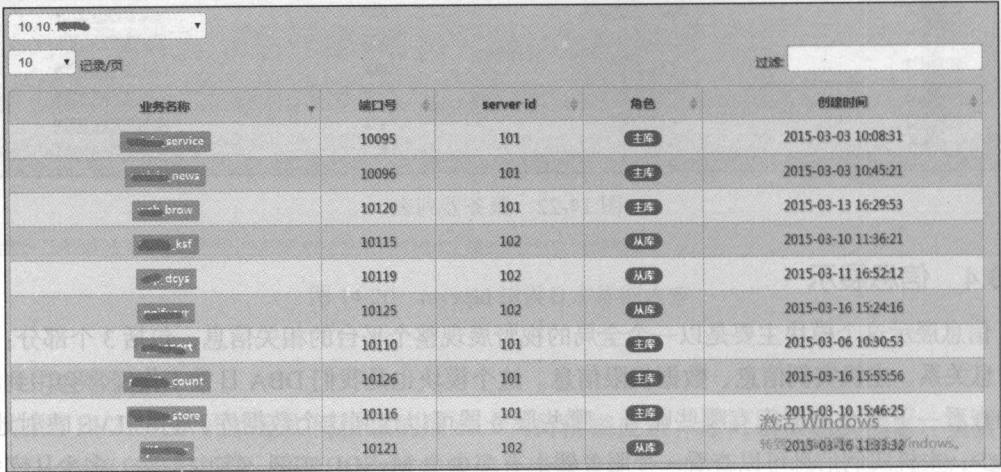
实例信息关系展示了每组实例的实例信息关系（主库、从库、备份库），LVS 映射关系，Redir 映射关系，账号相关，等等，可以下拉选择你想查看的库，也可以通过实例名或端口搜索，业务人员申请数据库时，我们就是从这里提取相关信息给他们的，如图 14-23 所示：

zabbix		搜索：实例名		GO	
主机详情					
主机	端口号	server id	角色	创建时间	
10.10.10.27	10392	103	备份库	2015-08-12 21:30:26	
10.10.10.29	10392	102	从库	2015-08-12 17:39:08	
10.10.10.27	10392	101	主库	2015-08-12 13:55:13	
LVS映射关系					
映射主机	映射端口号	lvs端口IP	lvs端口号	lvs主机	lvs组名
10.10.10.27	10392	10.10.10.27	10392	10.10.10.27	lb_lvs_B
10.10.10.29	10392	10.10.10.29	10392	10.10.10.29	lb_lvs_B
redir映射关系					
映射主机	映射端口号	redir主机	redir端口号	角色	
10.10.10.27	10392	10.10.10.27	10392	可读	
用户权限					
IP	端口	实例名	授权用户	授权ip	密码
10.10.10.27	10392	zabbix	slave	10.10.10.27	REPLICATION SLAVE
10.10.10.29	10392	zabbix	slave	10.10.10.29	REPLICATION SLAVE
10.10.10.27	10392	zabbix	zabbix_pro	10.10.10.27	ALL
10.10.10.27	10392	zabbix	zabbix_sel	10.10.10.27	SELECT

图 14-23 实例信息关系

2. 主机实例信息

主机实例信息展示了每个 DB 服务器上有多少数据库实例，并且可以看到有多少主库，多少从库，当一个服务器发生了物理故障，我们需要迁移切换故障服务器上的所有 MySQL 实例，原则是先主库后从库，这个时候这个功能就很有用了，如图 14-24 所示：

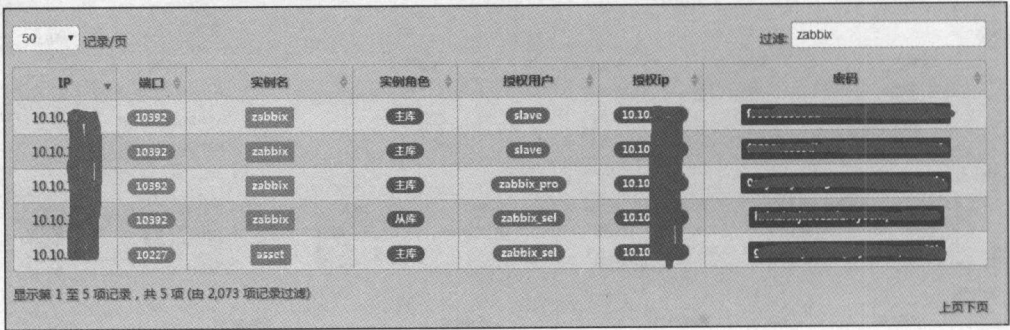


业务名称	端口号	server id	角色	创建时间
service	10095	101	主库	2015-03-03 10:08:31
news	10096	101	主库	2015-03-03 10:45:21
brow	10120	101	主库	2015-03-13 16:29:53
kaf	10115	102	从库	2015-03-10 11:36:21
deys	10119	102	从库	2015-03-11 16:52:12
	10125	102	从库	2015-03-16 15:24:16
rest	10110	101	主库	2015-03-06 10:30:53
count	10126	101	主库	2015-03-16 15:55:56
store	10116	101	主库	2015-03-10 15:46:25
	10121	102	从库	2015-03-16 11:33:32

图 14-24 主机实例信息

3. 数据库权限信息

数据库权限信息这个模块展示了整个平台所有的 DB 服务器对应的账号信息，可以查看某个账号在哪个 DB 服务器上有授权，还可以根据实例名查看该实例下有哪些账号，等等，如图 14-25 所示：



IP	端口	实例名	实例角色	授权用户	授权ip	密码
10.10.10.10	10392	zabbix	主库	slave	10.10.10.10	12345678
10.10.10.10	10392	zabbix	主库	slave	10.10.10.10	12345678
10.10.10.10	10392	zabbix	主库	zabbix_pro	10.10.10.10	12345678
10.10.10.10	10392	zabbix	从库	zabbix_sel	10.10.10.10	12345678
10.10.10.10	10227	asset	主库	zabbix_sel	10.10.10.10	12345678

显示第 1 至 5 项记录，共 5 项 (由 2,073 项记录过滤)

上页 下页

图 14-25 权限信息

14.3.5 入口 (LVS/Redir)

本节主要是介绍入口相关的功能，LVS 是程序访问 DB 的入口，Redir 是办公网访问 DB 的 NAT 映射入口，也就是说我们的真实 DB 地址是不对外暴露的，下面来具体介绍入口的相关功能。

## 1. LVS

LVS 是整个数据库平台的入口，其重要性不言而喻，所以我们会有多套 LVS（一套 LVS 有一个主节点和备节点），新建数据库的时候可以选择用哪套 LVS，从这里可以查看平台有多少套 LVS，哪些库用的哪套 LVS，对应的读写 VIP 是多少，还可以停止、打开某一条 LVS 转发，还有就是可以把某个库的 LVS 转发从一套 LVS 迁移到另一套 LVS，等等。

如图 14-26 所示，从“查看列表”选项区域可以临时断开一个映射，比如 zabbix 这个库配置了读写分离，而其中有一个从库有延时，我们需要从线上将其剔除，这时就可以选择从库的 LVS 映射，单击“open”按钮即可停止 LVS 转发；迁移 LVS 映射这个功能是为了防止一套 LVS 的主节点和备份节点全部不可用而设计的，不过这种情况发生的概率很低，除非是整个 IDC 挂掉或内网的出口交换机出现了问题。

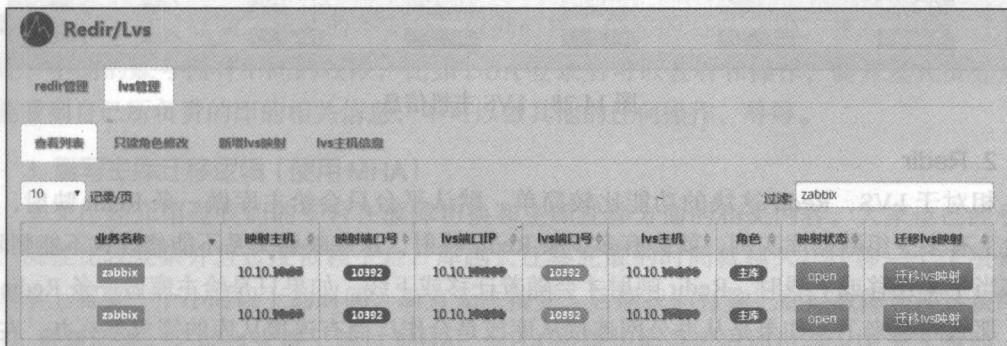


图 14-26 LVS 管理

一组数据库，新建的时候如果没有开启读写分离的功能，LVS 的读 VIP 默认是转发到主库上的，如果后续需要开启读写分离，则可以从“只读角色修改”这里添加从库，然后把主库从只读列表中删除，当然你也可以不删，这样主库除了承担写业务，也分流了一半的读业务，如图 14-27 所示：

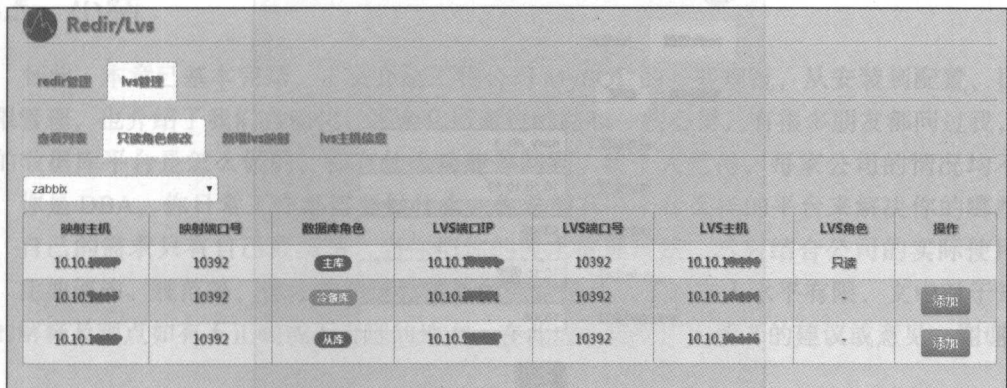
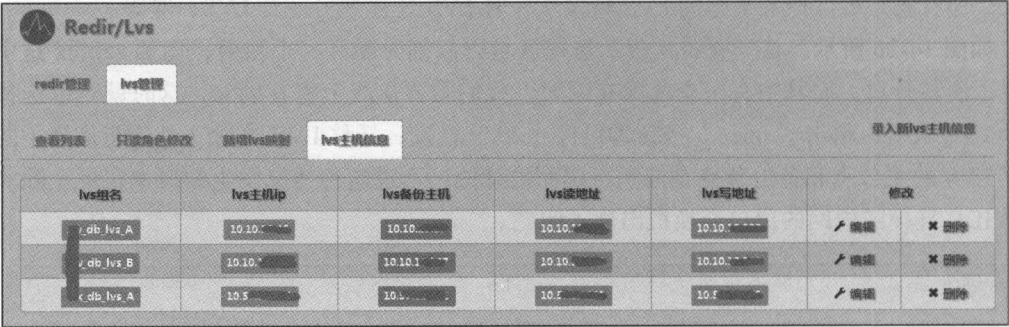


图 14-27 只读角色修改



LVS 主机信息就是管理 LVS 调度主机的，从这里可以新增、删除一组 LVS 调度，添加到平台之前，需要把 LVS+keepalived 的基础环境配置好，平台不负责这部分工作，并且也不去判断你的环境是否正常，也就是说 LVS 的环境需要你通过其他方法来保证它的可用性，如图 14-28 所示：



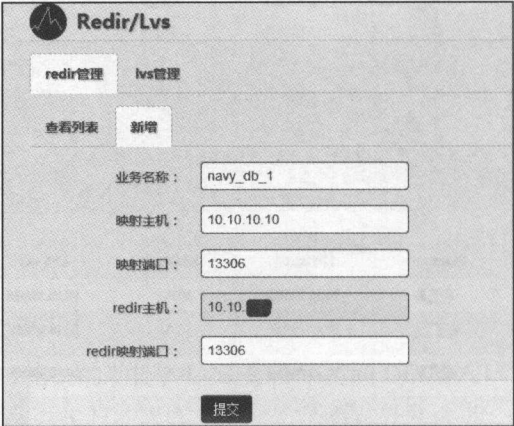
lvs组名	lvs主机ip	lvs备份主机	lvs读地址	lvs写地址	修改
db_lvs_A	10.10.10.1	10.10.10.2	10.10.10.1	10.10.10.2	<a href="#">编辑</a> <a href="#">删除</a>
db_lvs_B	10.10.10.3	10.10.10.4	10.10.10.3	10.10.10.4	<a href="#">编辑</a> <a href="#">删除</a>
db_lvs_A	10.5	10.1	10.f	10.f	<a href="#">编辑</a> <a href="#">删除</a>

图 14-28 LVS 主机信息

2. Redir

相对于 LVS，Redir 这块的功能比较简单，默认平台只会给主库做一条 Redir 映射，即平台上有多少组数据库实例，就会有多少条 Redir 映射，这些映射关系不能修改也不能删除，只有当主库迁移或下线时，Redir 映射才会随着迁移或下线。如果只是给主库做一条 Redir 映射，那么问题来了，如果是从办公网连接从库或是合作厂商有连接从库的需求怎么办，在这里我们也想到了这个问题，并且也做了新增 Redir 映射的功能。

业务名称就是库名，即实例名，映射主机就是需要映射的从库所在的 DB 服务器的 IP，映射端口就是这个从库的端口，Redir 主机的地址在平台上是固定的，因为它只是提供一个 nat 功能，基本上没有性能瓶颈，在可用性方面也是做的双机，底层有两台 Redir 主机，对外提供一个 VIP，Redir 映射端口就是对外提供的端口，如图 14-29 所示：



Redir/Lvs

redir管理 lvs管理

查看列表 新增

业务名称: navy\_db\_1

映射主机: 10.10.10.10

映射端口: 13306

redir主机: 10.10

redir映射端口: 13306

提交

图 14-29 Redir 映射

## 14.4 后期功能展望

至此，整个平台建设相关的内容和平台的功能已经基本介绍完毕，下面来说一下后期关于平台的重点规划。

### 1. 集成慢日志分析平台

关于 MySQL 的慢日志，我们是用 Flume+Elasticsearch+Kafka 编写的一个分析展示平台，后期会集成到数据库平台，并开放给开发人员，对慢 sql 最多的库进行全公司排名之类的活动，促进开发人员主动优化 sql。

### 2. 账号统一认证

目前平台登录所用的是一个本地账户，并且没有权限控制，相当于一个 admin，只有得到账号密码才可以登录，非常的不安全，后期会接入公司的 kssso 认证，登录必须通过 kssso 认证，不同的账号拥有不同的权限，比如 DBA 登录后可以查看和操作，而开发人员登录后只能看到自己所负责的库的相关信息，不可以做其他的任何操作，等等。

### 3. 重写主库迁移逻辑（使用 MHA）

上文已经介绍过我们的平台在故障切换和主库迁移方面的相关内容，大家也已经看到了，流程比较复杂并且过度依赖平台，库越大迁移完成的时间就越久，迁移一个 T 级别的库基本上需要一个小时以上的时间，简直无法接受，后期我们打算把底层故障切换这块结合 MHA 重新实现。之前也大概了解过 MHA，MHA 故障切换是可以通过漂 VIP 来实现的，即将 VIP 绑定在主库上，当主库不可用时，MHA 启动切换策略，选取一个从库提升为主库，并尽可能地通过 SSH 获取补全故障主库的 binlog，然后把 VIP 漂移到新的主库上，这与我们平台现在的单 VIP、多端口这种架构的差别有点大，如何把 MHA 和平台现有的架构很好地结合起来，对我们来说是一个很大的挑战。

## 14.5 小结

到此，本章已基本完结，本文介绍了我公司 MySQL 的一些规范，从安装到配置，再到权限管理，也介绍了我们自动化、平台化所走过的路和一些心得，有很多朋友都问过我，你们的数据库平台是怎么做的，都有什么功能等问题。我个人觉得，每家公司的情况均不一样，你是 DBA，你日常工作都需要做什么，你希望有一个什么样的平台来解决你的哪些痛点，自己的需求只有自己最清楚，当你把你的需求梳理清楚，然后结合公司的实际使用情况，比如架构、规范等，你才能知道你需要的究竟是什么了。本人水平有限，文中对于知识点的解释及观点如有不正确或不合理的地方，在此虚心接受广大读者的建议或意见，谢谢。



## 求职者与面试官

### 作者简介

赵昱, RHCA/RHCSS/MCITP, 熟悉 x86 平台基础架构系统的建设、管理及运维工作, 现就职于京东金融——网银在线, 担任支付产品技术部高级系统工程师一职; 9 年以上互联网金融、电信、政府等多领域背景的从业资历, 千万级大型项目经验, 优秀的文档撰写能力及沟通技巧; 曾参与中国国家电子政务多项重点工程的安全信任体系建设工作, 曾为中移动、国航等大型企业提供运维技术支持; 喜欢从事新技术研究、优化方案等工作。工作前倾向于制定有计划且详细的执行方案, 擅长处理远期规划设计类事件。善于在工作实践中分析问题、总结经验, 属于那种改进优化能力类型的工作者。致力于对 IT 技术的精研和业务分析, 帮助企业定制更加实际高效的解决方案。

我曾在两家公司担任过面试官一职, 在三年多的面试经历中参与过上百场的候选人面试。与传统的 HR 面试经类型的文章不同, 我将以求职者和面试官的双重身份, 以双方互视的独特视角, 在为读者深度剖析问题的同时, 与读者分享自己对职业发展和面试求职的一些观点与心得。

每个企业都渴望招到优秀的人才, 而进入一家知名公司工作也是每个求职者追寻的目标。然而理想和现实似乎总是有着很大的落差, 一方面, 企业找不到合适的人选, 另一方面求职者也总是在面试当中失利。这究竟是什么原因呢? 我想这个问题是值得大家去反思的。多年来我以求职者或面试官的身份参加过很多知名企业的面试, 在两家公司有过大约三年的面试官经历。我虽然不是 HR, 无法对面试进行专业的透彻分析, 但却可以从技术人员的别样角度去观察这两种关系微妙的角色。

我是一个善于在工作实践中分析问题、总结经验的人，属于那种改进优化能力类型的工作者。多年来我一直保持着一个好习惯，那就是每完成一项工作之后，都会反思和自省。每一轮面试下来，我都会回忆每一个环节的全过程，去思索哪里值得肯定，哪里做得不好。

那么作为求职者，如何做好职业规划？如何撰写一份吸引人的简历呢？哪些问题会导致面试失败呢？是否做好了加入一个新团队的准备？作为面试官，如何发布招聘启事，如何识别合适的求职者？面试官又是否树立了正确的面试观？哪些糟糕的表现会引起求职者的反感与抵触情绪呢？

正所谓“不知庐山真面目，只缘身在此山中”。只有跳出圈外，站在对方的角度考虑，才能发现很多意想不到的细节问题。当然凡事都有两面性，人们对世界的认知和理解，都深受自身生存环境与经历的影响，不可能产生完全一致的观点。所以在这里我想强调两点：第一，这不是一篇教程，没有对错之分；第二，本文所讲述的案例均来自于多个不同场景的描述，不针对任何企业或个人，请广大读者切勿对号入座。

## A.1 求职者篇

### A.1.1 职业规划从入职开始

说到职业规划，可能是很多人非常困惑的一个问题。很多人其实都缺乏这方面的意识，根本没有认真地考虑过自己的职业规划。他们希望参考别人的意见来确定自己未来要走的道路，或者认为应当先发展几年后才能定位，这些观念都是错误的。职业规划是一项自始至终都在进行的动态工作，在进入职场前就需要开始准备了。对于普通人来讲，职业收入是你的重要经济来源，关系到生活尊严与家庭幸福。怎么能够不认真对待？怎么能够依赖别人来指明道路呢？职业发展和婚姻同等重要，都是人生大事，存亡之道，不可不察啊。

传统相声《小神仙》里面有这样一个片段：讲的是一个人丢了东西，找了一位失明的先生给他占算，祈求能够找回失物。逗哏的最后就讽刺说，“你睁着眼睛都找不着，他能给你找得回来么？”虽说是个笑话，但在生活中，此类现象却比比皆是。我发现很多刚入行的新人总喜欢问这样的问题：A 职业和 B 职业，我到底应该选择哪一个？

作为新人，对职业的选择有困惑是正常的，但不应该盲目地跟从他人的建议，更不可让他人为你设计什么发展道路。正确的做法应当是：先向业内人士了解每个行业的发展趋势、职业特点和技能要求，再结合自身的性格和优缺点来选择发展道路。请记住，不管走什么路都一定要自己去决断。因为只有你最了解自己。如果你迷惘到连自己都不了解，那么别人指出的道路又怎么能够确保就一定是正确的呢？一句话，你自己就是你最好的职业规划师。

当然只具备意识是远远不够的，还需要明确的计划和切实的行动。

有一个小伙子参加我的面试，基础知识很差。对于新人，我并不是十分计较技术方面的落差。我问他一个问题：如果你加入团队，愿意做多久？有什么职业规划？他满怀激情地

告诉我，要在一年内成为团队的核心，五年内成为架构师，第十年成为技术总监。我继续提问：团队核心需要掌握哪些技能？你打算如何实现？此时，他沉默了。从他脸上不自信的表情可以看得出，他对自己的目标定位和实现方法根本就不了解。职业规划不是喊口号，和商业企划一样，要有详尽的方案才行。这项工作至少要分成这几个部分：定位目标、找出差距、设定计划、付诸实施。

俗话说“男怕入错行，女怕嫁错郎”。制定一个长远的职业发展方向是非常有必要的。请注意，这里只是确定职业发展方向，不需要过于具体化。我从事 IT 行业近十年，在这个过程中始终思考着自己的职业发展，分析自己的不足之处并加以改进，不断地修正自己所走过的道路。就像松下幸之助的 250 年计划无法帮助松下走出困境一样，我们同样也不可能一开始就看得那么远，那么准确。只有随着经验和阅历的不断提升，你才能够深入地认知自己，正确地解读行业。这时候，位于你前方的发展道路才会逐渐地清晰起来。

在确立了大方向之后，你还需要制定几个短期的目标作为支撑。不用担心它们和你的终极目标是否匹配。因为以你现有的能力距离实现终极目标的要求还差得很远，所以不管做什么都不会是无用功。比如你想到南极洲去看企鹅，不管你是造飞机舰船，还是买羽绒服，或者干脆去挣钱，其实都是对的。当你完成了这些短期目标之后，就可以依靠这些新技能去进一步发展。短期目标的制定一定要切实可行，目标太多或难度太大都是错误的。

然后你要搞清楚，现有条件与目标实现所需要的条件这两者之间的差距是什么？正所谓“知己知彼，百战不殆”。你需要制定实现目标的详细计划、时间进度表和最终验收标准。你可以不断地对目标进行细化和分解。使用“脑图软件”是一个不错的选择，它可以帮助你把一个目标拆解成多个小任务。你可以计划每周去完成一个小任务。关于时间进度表可以使用微软的 Project 软件去做 WBS（Work Breakdown Structure，工作分解结构）。对于新人，可以求助于有经验的前辈帮助你进行完善。

最后就是 Just do it——付诸行动。在这个过程中，你要不断地审视工作进度和执行效果，有问题及时修正。所谓做比说强，聊胜于无。相比那些夸夸其谈的“立志人”，我更欣赏行动迅速的“践行者”。要知道，事情只要做起来，那么你就已经开始在通往成功的道路上前行了。

### A.1.2 如何写简历

作为面试官，第一步工作就是简历筛选。一般我会花费两分钟左右的时间去阅读一份简历并加以评估。下面这种简历的内容形式和排列顺序是我比较推荐的。

- 求职意向。
- 个人基本信息（涵盖从业时间，行业背景的描述）。
- 个人简介（涵盖对技术、管理、优势等情况的高度汇总）。
- 工作经历（时间按照倒序排列，将主要的工作业绩以数据的形式体现出来）。

一般求职网站在填写工作经历时，都是以公司作为划分单位的，如图 A-1 所示。我认

为这种划分方式过于粗犷。因为你在这家公司的所有业绩都只能填写在同一个表单之中，而表单的字数是有限制的。有些求职者长期效力于一家公司，成绩斐然。如果因为受到字数限制而影响了自身展示那将是不公平的。另外这种方式缺乏细粒度的时间划分，只有工作量没有时间点，也是很大的弊病。因为这些成绩也许都是前期产生的，而后期可能只是业绩平平，所以很难反映出求职者真正的工作状态。在众多求职网站当中，我最喜欢的就是猎聘网。猎聘网在创建简历时，针对每一家公司，会有一个“添加本公司其他任职”的按钮，可以根据时间和职位进行二次分类，如图 A-2 所示。这就很好地解决了上述问题。如果求职者在这家公司有职位晋升的情况，还可以据此观察求职者的职业发展历程。目前很少有求职网站效仿这种做法，该项功能对于在一家公司服务期超过五年的求职者来说是非常有必要的。

求职意向		修改
期望工作性质：	全职	
期望从事职业：	软件/互联网开发/系统集成	
期望从事行业：	互联网/电子商务	
期望工作地区：	北京	
期望月薪：	25000元/月以上	
目前状况：	我目前处于离职状态，可立即上岗	
工作经验		修改
2016/01 -- 至今：	第二家公司   系统工程师 互联网/电子商务   25000元/月以上 XXXXX XXXXX XXXXX XXXXX	
2015/01 -- 至今：	第一家公司   系统工程师 互联网/电子商务 XXXXX XXXXX XXXXX XXXXX	
教育背景		修改
2000/09 -- 2004/06：西工业大学   信息管理与信息系统   本科   统招		

图 A-1 其他网站的模板

工作经历		新增
2015.01 - 至今 某某公司		
IT服务/系统集成		
高级系统工程师	2016.01 - 至今	
工作地点：北京   下属人数：0		
工作职责：YYYY YYYY		
初级系统工程师	2015.01 - 2015.12	
工作地点：北京   下属人数：0		
工作职责：XXXX XXXX		
+ 添加本公司其他任职		

图 A-2 猎聘网的模板

### A.1.3 面试前的观察与考量

首先我要在这里纠正一个错误的观点。面试不是企业对求职者单方面的考核，而是一次彼此都要接受检验的双向选择。当然，我们不得不承认，在面试中求职者往往处于不对等的弱势一方。但求职者在收到 offer 之后，同样也有反向选择的权利。然而是否要加入一家公司，也许不一定要在收到 offer 之后才决定，在一开始面试之前，你就可以有所观察。

不管这家公司的知名度如何，你最好都要先去了解一下它的背景和基本情况，包括行业地位、发展趋势、企业文化、业界评论等。另外，办公地点也要事先了解清楚。我个人认为如果有可能的话，最好还是能够进一家大中型企业。一个刚入行的新人就像一张白纸，第一家公司会对他们产生比较大的影响。大公司做事情相对比较规范，对新人的影响还是比较积极的。其次也可以考虑创业公司，相对于小公司，创业公司也许能学到更多的东西。但是关于创业公司，我只建议工作不足两年的新人去考虑。这个问题我会在后文中详细分析。总而言之，我们更愿意投身于一个做事规范，积极向上，有发展前途的公司。

在正式进入面试环节之前，你最好留意一下这家公司的办公环境、前台员工的职业形象及待人接物，等等。如果路过办公室，还可以观察一下公司员工的工作状态。他们是快乐奋进的？是疲惫痛苦的？还是百无聊赖的？员工的形象和工作状态是我判断一家公司优劣的基本依据。

另外，我会计算一下我的等待时间。因为等待时间是和重视程度成反比的。迟到是求职者的大忌，但有些面试官反而却不够守时，让候选人等待很久，实在是非常的不妥。虽然运维是个比较特殊的行业，但既然是事先约好的会晤就应该做好安排。除非真有特殊情况，否则没有理由让候选人等太久。如果技术环节的面试官迟到并告知你是因为在处理突发故障，那么有经验的候选人会就此发现一个问题，要面试官去处理临时故障，可能存在两种原因：①这个岗位严重缺人，目前可能只有这位面试官一个人；②这个团队的技术能力有问题，其他成员无法很好地胜任这个岗位的工作，表明团队成员技术落差大，人员存在单点。接下来你可能就要多留心了。你一定要搞清楚这种情况已经持续了多久。如果长期没有得到改善，则说明加入这个团队将会有很大的风险。

在反向提问的环节，求职者还应该更多地了解工作环境、团队情况、职位发展等问题。例如可以提出如下一些问题：

- ❑ 请问我们最近两年的发展目标是什么？要做到怎样的程度？
- ❑ 我能了解一下我们团队近两年的发展规模是怎样的么？未来我们还需要哪些成员加盟？
- ❑ 请问您认为目前我们团队所面临的最大需求是什么？
- ❑ 请问您最欣赏的员工应该是怎样的？
- ❑ 请问目前我应聘的这个岗位未来三年的发展方向是什么？如果该岗位能够进一步发展，还需要哪些要求呢？



### A.1.4 候选人为何面试失利

#### 1. 搞不清状况

作为面试官，我遇到过两次候选人向我询问薪酬问题的情况。首先我只负责技术考核，后面还有总监和 HR 的环节。和我谈薪水既不合时宜，恐怕也找错了对象。尤其是在候选人还没有获得认可的情况下，此举无异于自杀。几乎所有的面试官，尤其是 HR 的 MM 对此是非常反感的。我想提示所有的求职者：如果面试官在交谈中主动和你谈及薪酬问题，说明你已经成功了 90%，但要是你自己率先触碰这个敏感话题，那么肯定不会有什么好结果。

#### 2. 狐假虎威的个人介绍

这类候选人在描述工作经历时总喜欢把主语设定成“我们”。例如，我们团队在某项目中开发了 ×× 软件，建设了 ×× 平台，完成了 ×× 工作等。请注意，这是面试而非招标。我希望了解候选人究竟做出了哪些成绩，而不是他的团队和公司。我在研读了一些心理学的书籍之后，通过对面试者面部表情的观察发现：这些候选人，在使用“我们”作为主语时言语激扬，显得比较兴奋。而当他们被要求只描述自己的工作业绩时，也就是把主语更换成“我”时，立刻就会显得底气不足，同时语速下降，眼神也开始变得黯淡。

#### 3. 基于私有化产品的工作经历

我经常会在简历里面看到关于私有化产品的描述。对于私有化产品，如果你做的是私有化产品研发，我会觉得很不错。但如果只是私有化产品运维，我希望对方能更加了解底层通用的技术，而非简单的手册操作。

#### 4. 长期重复性基础工作，技术深度不足

我见过很多工作经历超过七年的做产品实施类的工程师。他们确实接触过很多主流的软硬件产品，但其水平只停留在 ICM 层级。ICM 这个名词来源于 VMware 的认证课程，是 Install Configuration & Management 的缩写。他们的任务就是完成产品安装配置的交付工作。他们接触过的产品非常多，但对其深层次的技术并不十分了解，自身更是缺乏大规模运维的实践经验。我对他们多年来一直只从事着一项重复性的劳动深感忧虑和惋惜。此类工作毫无技术壁垒，很容易就会被用工成本更低的人所接替。

#### 5. 动力不足，执行力差

工作意愿和学习动力不足，或者是执行力差的候选人，即便技术环节勉强过关，最终也无法通过部门总监的面试。没有工作热情、缺乏输出效率的人，就像是一只大电阻，白白地消耗着能源，有效功率却很低。聘请这样的人来工作，不但带来不了多少价值，而且这种坏风气还会在团队里传染开来，影响士气。这类候选人通常都是来混日子和寻开心的，他们的工作目的只是为了进入一个带光环的平台或是更换一个安逸的环境而已，我在他们眼中看不到进取的目光，更多的是迷茫和麻木，言谈话语中已经对运维工作没有了激情。

**【案例分享】**C 君，从业三年，传统行业，小型机工程师，希望到互联网行业发展。



面试官：你接触过哪些开源的产品？

候选人：主要关注过 OpenStack 的一些技术。

接下来面试官就 OpenStack 的体系架构及重要模块的工作实现方式等对候选人进行了技术考核，发现对方对此知之甚少。

面试官：你从什么时候有了想进入互联网的想法？

候选人：大概是半年之前有了这样的想法。

面试官：你现在所从事的工作，每天的工作量是多少？

候选人：现在的工作不是很忙碌，每天大概有两三个小时左右的空闲时间。

**【案例点评】**到此为止，我想候选人已经把他最后的机会都浪费掉了。足足半年，每天两到三个小时的空余时间，如果认真学习应该也已经小有成绩了，所以我是不太看好那些光说不练的人。

## 6. 可靠性差，根本不值得信任的人

打网游的朋友都知道一句话：“不怕强大的对手，就怕坑人的队友。”就像没人愿意用一个可靠性差的软件一样，与不可靠的人共事最终会毁了整个团队。下面这些人都是团队的大忌。

☐ 做事随意性强、不遵守已有规章制度，工作缺乏规范性的人。

☐ 凡事凭直觉做，缺乏细致分析，考虑问题不周全的人。

☐ 工作不认真，喜欢糊弄，抱有“60分万岁”主义的人。

☐ 责任心差，做事毛躁，对生产环境缺乏敬畏心的人。

☐ 自我意识太强，无法融入团队的人。

☐ 执行力差，工作效率低下，光说不练的人，

☐ 缺乏担当和大局观的人。

☐ 不喜欢做事，整天只会抱怨或到处搬弄是非的人。

**【案例分享】**某公司招聘部门经理，目前只有两名候选人。

S 君，从业八年，效力过四家公司，现任职某公司部门经理。

公司	任职（年）	担任职务
4th 0.5	部门经理岗	（有带团队经验）
3rd 2.5	高级工程师	（有带团队经验）
2nd 2.5	高级工程师	
1st 2.5	中级工程师	

L 君，从业八年，效力过四家公司，现任职某公司资深工程师。

公司	任职（年）	担任职务
4th 3.5	资深工程师	（有带团队经验）
3rd 2.5	高级工程师	（有带团队经验）
2nd 1.5	中级工程师	

### 1st 0.5 初级工程师

从他们所做过的工作上来看,这两位的技术能力都很强。S君还具备半年左右部门经理的工作经验,而且这两位都是猎头公司推荐的人选,那么作为面试官你会邀请谁参加面试呢?

【案例分析】最终,S君在简历环节被面试官淘汰了。一个高级职位除了对技术和管理能力有较高的要求以外,更加看重的是候选人的责任心。一个部门的负责人要有担当,缺乏稳定性是不能容忍的。既然是猎头推荐,说明候选人是有离职意向的。这两个人平均一家公司的就职时间都在两年左右,从这一点上来看,应该都是不合格的。

不过L君最近一家就职时间已经三年多了,并且从历史时间上看,离职频率是呈下降趋势的。他之前晋升的速度非常快,说明他的学习能力比较强。而带团队的经验相较S君也更加丰富一些,有一定做管理的基础。综合考虑他几次离职有可能是碰到了职位天花板的缘故,但至少是值得约见的。当然在面试过程中也需要仔细考察衡量该候选人的职业发展意向、岗位稳定性、期望值要求及目前公司所能提供的条件等诸多方面因素。

但是S君就不同了。基本上是两年左右触发一次离职事件。频繁的离职表明了对对方缺乏责任感,这种朝三暮四的人难以获得别人的信任。他作为部门机构的负责人,整个团队的领导者,刚刚半年就出现了离职的倾向。综合来看有两种可能:第一种,他目前所处的部门出现了一些棘手的问题,导致团队陷入了困境,个人承受不住压力,考虑离开。第二种,他在面对更好的机会时,就产生了放弃现有团队的想法,缺乏职业道德。不管是哪一种,这种遇到困难或利益就没有了担当的人在进入团队的第一天,就将是灾祸的开始。

【案例点评】京东创始人刘强东曾经把人分为4种:第一种是金,即能力强又符合公司价值观的人才。第二种是铁,即符合公司价值观但是能力较差的人,通过锤炼锻造可以成为钢。第三种是废铁,这种两者都不具备的人在面试时就会被直接淘汰,不足为虑。最可怕的是铁锈,即能力极强但是不符合公司的价值观。这种人非常危险,因为他们不但难以识别,而且破坏力惊人,不及时清除会腐蚀掉整个团队。

我大概在五六年前也总结过一个类似的观点:“一个人的价值所在是一种矢量的体现,能力大小只决定其长度,但人品决定了最终方向,也决定了价值的正负。”假如在街上有两个恶棍正在持械行凶,一个持刀,另一个持枪,你认为哪一个更危险呢?尽管我是技术环节的面试官,但我始终把技术放在第二位。技术差一点可以学,就像老刘所讲的通过锤炼可以改变。但是如果技术非常强,而人又有问题,那么这种人的技术能力和他即将带来的破坏力是成正比的。

## A.1.5 如何突破职业发展瓶颈

应该说,职业发展瓶颈(图A-3)是一个非常普遍的现象。因为优秀的岗位总数量是很少的。很多同行在工作了几年之后,都会出现类似的情况:发现自己在某个岗位上长期无法突破,工作内容没有什么本质上的变化,薪酬待遇也总是停滞不前,职业竞争力开始下降,

同时也产生了倦怠的工作情绪。

那么问题症结在哪里呢？我认为首先要承认自身能力出现了瓶颈，这也是根本原因。如果你无法认识到这一点，那么你将在错误的道路上越走越远。很多人发现薪酬待遇无法提升时，往往会在自身没有做出改变的情况下，就想通过更换新的工作来改变目前的窘境，这是一种非常不明智的选择。

这就好比这样一则寓言故事：西村的树林里住着一只猫头鹰。因为人们讨厌它难听的叫声，总是想方设法要赶走它。于是，猫头鹰决定搬到东村去。半路上，它遇到了一只斑鸠。斑鸠问：“你这是要飞去哪里？”猫头鹰叹了口气说：“西村的人嫌我声音难听，都讨厌我，我决定要搬到东村去。”斑鸠听后笑了起来：“老兄，依我看，如果你不能改变你的声音，不管你搬到哪里去，都会是一样的结果。”

当然，我不否认同等岗位的薪酬在行业内是有一定差异的。但如果你认识不到自身能力的不足，不去主动做出改变，转换平台也无益于根本问题的解决。即便上天真的给了你一个高薪的职位，你自身无法承载相应的要求，最终也只能放弃。我想让大家明白一个道理：**做事情，改变目标不如改变方式，改变环境不如改变自己。**

图 A-3 简要概括了职业发展的瓶颈：

我总结了几种类型的人，他们往往很容易就出现职业发展瓶颈。

### 1. 原地踏步型

这是最典型的职业瓶颈的体现。这类人长年累月重复着某一类工作，而且已经形成了一种惯性的工作模式，因为某些原因，他们无法或不愿意做出改变。

### 2. 浅尝辄止型

这也是一类典型的现象，这类人接触过很多领域的知识，但可惜的是对每一样都不够深入。

他们喜欢对每种软硬件产品都进行尝试，当发现用不下去时就尝试着变换另一种产品，对产品的实现方式和体系架构都不甚了解，最终沦为了产品的试用者和宣传员。

### 3. 挖深井型

这一类人钻研的领域往往应用面比较窄，可以说是前一种类型的反向极端。

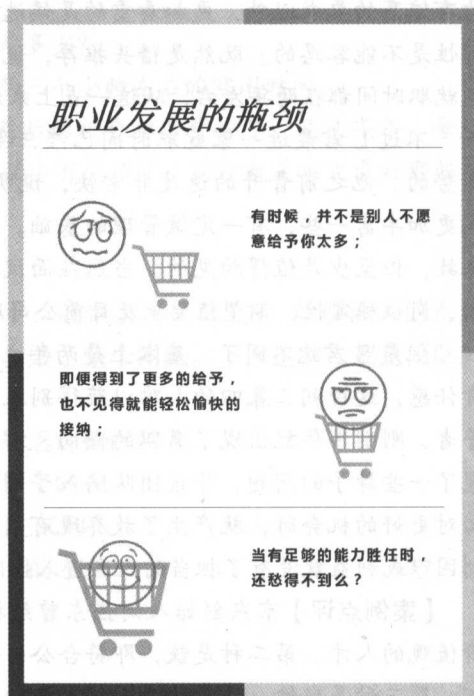


图 A-3 职业发展的瓶颈

【案例分享】我曾经面试过一名在一家互联网企业做硬件测试的工程师，工作经历大约六到七年，一直在从事 Bench Mark（IT 行业中的 Bench Mark 是指：依照特定的标准，使用专业的工具，针对同种类型的软硬件产品进行性能检测，最后依据检测结果的比对，对产品的优劣做出评价）的工作，并且负责一个小团队。这位候选人对目前的薪水不够满意，但拒绝了一家知名硬件厂商的邀请。因为他希望继续留在互联网行业并从事原有的工作。可以说他在硬件测试方面确实是个行家，但是他似乎比较排斥学习他认为没用的其他领域的知识。因此他最终也没能通过我们的考核。

我们给出的落选理由如下：第一，Bench Mark 可以为服务器采购大单节约可观的资金，而且节省成本的 5% 就足以支付这位工程师的薪水了，但是 Bench Mark 的下一个有效周期过长，在接下来一段时间内无法继续产生利益，虽然很重要但完全可以找人兼职完成。第二严重缺乏其他领域的知识，对职业发展产生了不利影响，难以胜任更高的管理或技术岗位。我个人认为这位候选人如果不愿意改变看法的话，拒绝那家厂商的邀请很显然是十分可惜的。

可见，挖深井是有极大风险的。除非你研究的领域是垄断性的或极具市场需求，否则很可能这口深井就是葬送自己职业生涯的坟墓。

职业发展最好是着重一个方向，在不断深入的同时也注意横向知识的积累。当达到一定核心深度时你会发现各领域之间的界限已经开始模糊了。专注一个方向发展并不会影响到你的知识拓展，相反这种学习方式也会使得广度学习更加深入且有针对性和针对性。也就是我们常说的 T 型人才，但我个人更愿意称其为扇形人才。切记不要做“万金油”，什么都会就是什么都不会。

## A.1.6 如何实现自身能力的扩展

### 1. 拒绝重复性劳动，日常工作工具化

在这里我认为不适宜讲自动化的概念，我认为现今自动化概念的门槛被降得太低了。不是说写个脚本或程序就能称为自动化运维了。如果把手动操作比作走路的话，自动化应该是无人驾驶的汽车才对。目前我们也只是做到了工具化的级别，最多就是自行车而已，仍旧需要大量的人工干预环节。事实上，能做出尽量减少人工干预的工具化产品，也是一件非常不容易的事情。

### 2. 提升工作效率，多做第二时间象限的事情

我个人认为优秀团队不是看加班多少，而是看工作效率和有多少比例在做第二象限的事情，第二象限的完成目标应该在 PKI 中占有更高的比例才对，而不是欣欣然地看扩张速度，看团队成员为赶项目熬了多少个通宵。如果一件事情不做好品质，做得再快也只是产生了更多的廉价垃圾而已。

## 拓展：时间管理四象限的基本知识

第一象限：紧急重要，属于灾难型事务。出现这类事务往往会引发非常严重的后果，

需要立即解决。

第二象限：重要不紧急，属于基础架构类的规划设计型事务。该类事务的影响效力是长期的、全局性的，一旦形成决策，不可以随意修改，错误的决策将导致成本不可控的重构工作。如果此类事务被延误，迟早会演变成第一象限事务。

第三象限：紧急不重要，大部分属于取悦别人的服务型事务。通常来说不得不做，但要学会尽快处理完毕。第三象限事务往往大量存在于常态化工作中。因此，在技术层面上要考虑工具化处理，减少人工干预；在管理层面上要考虑优化规范、精简流程。

第四象限：无关紧要，纯属垃圾事务。这些事务往往与工作无关或属于无效冗余，应该最大化地杜绝它们的出现。

### 3. 审视自身目前的缺陷

如果认识不到自身的问题，就无法做出正确的调整，付出再多的努力也是白费的。

### 4. 同伙伴一起成长

在艰难的旅途上行进的时候，如果有志向相同的伙伴便不会感到孤独和困苦。出现问题的时候可以一同探讨解决而不是独自苦思冥想，尽快结识这样的伙伴，有时候比拜一个好师傅更重要。

### 5. 用心做事最重要

做事情有四个层次：做任务、做工作、做产品、做品牌。有些人做事就像挤牙膏，指使一样干一样，多一点儿都不带做的。还有些人做事好像机器人，遇到异常情况，既不会变通也不会提前汇报，继续按照原先的方案做，出了问题反倒责怪别人——“当初你没告诉我应该这么办啊”。与其聘用这样不用心的员工，还不如写个严谨的程序来做呢。要想有所成就，做事情不但要用心，更要有匠心。

欧米茄（Omega）是国际著名制表品牌，由路易士·勃兰特始创于1848年，拥有将近170年的悠久历史。其代表符号“Ω”是希腊文的最后一个字母。象征着事物的伊始与终极，代表了“完美、极致、卓越、成就”的非凡品质。作为世界上最具影响力的腕表品牌，曾缔造多项腕表精度纪录，它最具智慧之处就在于：没有任由工业化泛滥到制表的每一个环节，而是将腕表的最后一道工序保留手工完成，坚守着“拒绝批量生产，秉承手工制作”的瑞士制表业的传统。

但那些同为瑞士机芯的量产表，它们的命运却只能是成为地摊儿货。当前这个讲求高效高收益的社会，给人们带来的是急功近利与浮躁的心态，反而失去了本属于中国优秀传统文化的匠心精神。把事情做到极致、精益求精不过是匠人的最低标准，匠心精神还在于强调以下几个下方面

□ 坚持、专注：无论外界环境如何变化，匠人对做事原则和骨气都有着高度的坚守。

□ 谦恭、自省：匠人实事求是，谦逊虚心，不自大，不吹嘘。



□ 敬畏、入魂：匠人对工作有着崇高的职业自豪感与使命感，将自己的生命与灵魂注入到作品当中。

总而言之，缺少匠心的产品就没有艺术和价值，只有真正的匠人才值得人们尊敬。

图 A-4 概括了如何实现自身能力的扩展：

### A.1.7 是否为五斗米折腰

世界很小，圈子也很小。从业多年后，当你在行业内有了一定地位或声望的时候，难免会接到一些猎头或以前共事伙伴的邀请。而当你认为自己完全有能力胜任的时候，是否会考虑更换一个新的环境呢。面对所谓的诱惑，你又是否真的做好了接受的准备呢。

#### 1. 什么情况下考虑离开

我可以理解的离职原因只有三个：前途、家庭和工作环境。关于前途一般是团队平台影响到了个人职业发展的时候。前面讲过，职业发展瓶颈主要是基于自身的原因，但也有客观因素存在。比如一只超级球队，连替补都是一等一的球星。身为板凳队员并非不够优秀，而是优秀的人实在是太多了，而上场人数又有限。在职场也是一样，某人本来可以胜任更高的职位，但是已经有人占了位置，这个时候就只能选择离开，像这种情况就称为职位天花板。平台的机制问题（例如绩效考核、管理制度、团队建设等）也是导致人员流失的主要因素，但它与前者带有不可抗力的属性不同，这类问题是可以改善的。因此团队的领导者应当重视机制的建设和改善。

#### 2. 何为好的平台，什么样的团队值得加入

##### 1) GDP 与实际收益

很多新人刚开始会比较看重自己能学到多少东西，希望能加入那些拥有众多牛人团队的大平台。我想要说明一点的是：牛人其实在每一家公司都是必备的。也可以说这些牛人就是公司最主要的 GDP 来源，如果一个公司连这些都不具备那就离倒闭不远了。GDP 越高，意味着人均 GDP（你的成长机会）也就越高。但是大家都懂一个道理：人均 GDP 并不等于实际收益，它们完全就是两码事。

什么叫做实际收益呢？就是那些你真正能够获得的东西，实际收益不但要看得见，还要摸得着。关键不在于你的团队有多少牛人，而是有多少愿意和你一起分享，一同成长的伙

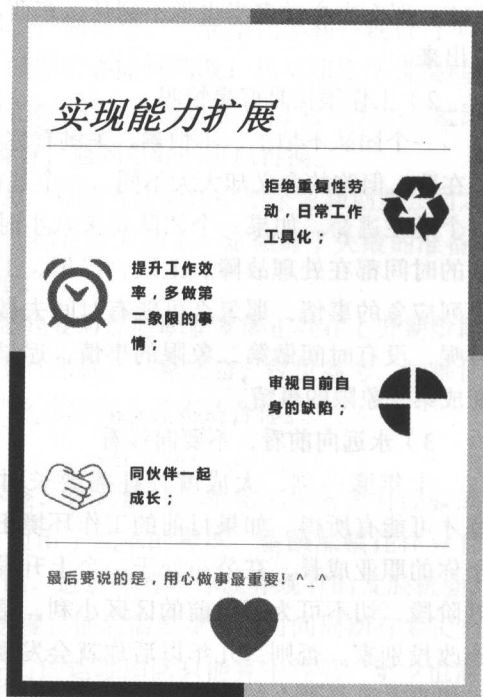


图 A-4 如何实现自身能力的扩展



伴。当然如果有一位大牛愿意带你成长带你飞，那当然是再好不过的了。

但是如果你与这些牛人在工作领域中没有任何交集，或者出于各种原因他们无法或是不愿与你分享，光有牛人却学不到东西，则是毫无意义的。那么能不能主动去学呢？没错，具备主动学习性是好的，但是如果团队成员无法自发地去做分享，或者所谓的分享知识只流于形式，那么再主动多半也学不到什么东西。毕竟公司不是学校，没法安排一个专门学习的岗位出来。

### 2) 工作很忙是好事情吗

一个团队不怕忙、不怕累，关键看你在忙什么。大家看下图 A-5 中的小人儿，左右两边都在跑，但跑的含义却大大不同。一个是在健身，一个是在逃命。如果一个团队每天八小时甚至更多的时间都在处理故障、上线、维护、工单等一系列应急的事情，那怎么可能会有时间去做技术提升呢，没有时间做第二象限的事情，迟早都会演变成第一象限的事情。

### 3) 永远向前看，不要向钱看

十年磨一剑，大成就往往需要长时间的磨练才可能有所得。如果目前的工作环境正好有助于你的职业成长，在公司处于一个上升发展的良好阶段，切不可为了眼前的区区小利，急急忙忙地改投别家。否则，几年以后你就会发现，新东家似乎并不是那么的重视你，而自己原有的发展机会也白白地让别人捡了便宜。大家都玩过跳棋吧？往前跳的棋子未必能走多远，但一定会为后面的棋子让开道路。离职有时候看似是自己的机遇，却往往是为他人做了嫁衣。

图 A-5 简要概括了何为好的平台：



图 A-5 何为好的平台

## 3. 高薪水是有代价的

没有金刚钻，别揽瓷器活儿。如果你参加完一个面试后，发现自己胜任这项工作尚有困难，对方却发来了薪水诱人的 offer，这时候最好是谨慎接受。

大家去肯德基就餐，会发现单点的东西总是比套餐要贵得多。再有比如可口可乐，330ml 的罐装饮料售价大约在 2 元左右，而 2L 的大瓶装虽然售价高达 6.5 元，但容量却是前者的 6 倍。在职场里也会出现同样的情况。往往收入越高其性价比越低。比如当薪酬是 1，此时的要求也是 1。而当薪酬是 10，此时的要求往往变成了 12 或 13。我将其称为“全家桶现象”。没有一个公司是出来做慈善的，高薪水往往意味着你要为此付出更多的代价。

企业有时候急于求成也难免会看走眼，请来了不合适的人选，结果弄得双方压力都非常的大。行业圈子本来就很小，如果承担不了压力被迫离职是件非常糟糕的事情，传扬出去反倒坏了名声。

#### 4. 是否应该投身创业公司

假如你身处荒郊野外，现在这里有一辆越野车、一辆马车、一辆摩托车和一辆自行车，你会选择搭乘哪一种交通工具呢？越野车固然好，但你未必能够驾驭；马车速度虽说慢些，也还算不错；摩托车车速快，但安全系数低，也许能最先到达终点，也有可能一不小心就滚下了山涧；自行车很辛苦，不过总比走着强，先骑着看，遇到更好的可以再换。

很多人都会面临这种选择。大公司不一定进得去，有些人就会在中型企业和创业公司之间犹豫。实现财务自由是创业公司最大的卖点，但选择它的人却不一定做好了失败的准备，也许这是因为他们距离第一次互联网革命的时代太远了。

关于是否加入创业公司这个问题，以下这些话是我要对那些职业发展正处在上升期阶段的同学们讲的。所举的例子，只为讨论分析之用，没有恶意。一家之言，仅供参考。如果你已经具备了高管的实力，或者是初入职场的新人，那么这些观点并不适合你。

##### 1) 30% 能买断几年的青春？

创业公司的特点是设备飞速扩张，团队人员不整，有大量的初始化工作需要经验的实施团队来完成，因此有很多事需要你从零做起。但出于成本的考虑，薪酬涨幅往往只有20%~30%。对于那些正处在上升期阶段的同学来说，这意味着要你放弃现有的发展机会，一下倒回解放前。那么重新来过到再次追赶上现有进度，前后需要花费的时间周期有多长？你是否对此进行过评估呢？不说别的，一个60人左右的运维团队只能算中等吧？至少也需要两三年左右的时间才能找到合适的人选。人家用了30%的钱，就买断了你至少两三年的青春。而你有没有想过，两三年以后你和以前小伙伴的差距会有多大？所以，不要单纯地去看薪酬，要从多个维度去权衡利弊。

##### 2) 财务自由——不要让赌徒心理作怪

如果说30%不能撼动你的话，财务自由也是一个令人神往的理由啊。不过提醒大家的是，创业本身就有一定的风险。但实际上有些人是抱着押宝的心态去创业公司的。有这种想法是非常危险的，这种有着赌徒心态的人不管到哪里都是不会获得成功的。另外，我还要再给大家泼个凉水。宋江和卢俊义上梁山比很多人都要晚，但这两个人却依旧坐了头两把交椅。我的意思是：如果你成为不了团队的高层与核心，恐怕财务自由这句话也不是对你说的。

##### 3) 成功没有充要条件

牛人云集也好，资金雄厚也罢，这些都不能成为保本儿保息的有力依据。当年十八家诸侯讨伐董卓，有谁能确保自己的团队肯定就能成就霸业呢？刘备文有卧龙凤雏，武有五虎上将，坐拥西川，最后七十万大军还不是兵败于白帝城吗？猪八戒好吃懒做，工作能力差，还

总在团队里散播负能量，但始终没离开取经队伍，最后他也修成了正果。有时候站队就是一种随机选择，真就没什么道理可讲。当下市场竞争，没有谁是绝对安全的。成功是受多种主客观因素共同影响的。首先自己要努力，同时还需要有适当的机遇来辅助，客观条件是可控的，所以要摆正心态才对。

#### 4) 要弄清楚对方是求贤若渴还是求职若渴

要了解清楚，人家到底是需要你这个人，还是你这个角色。如果只是要你来撑场面救火的，一旦人员齐整后，你是否会被后来者居上呢？不要甘当了铺路泥，到时候连铺路石都算不上。

#### 5) 愿赌服输也是需要资本的

有一个笑话：有一个人和一只鹦鹉同坐飞机。在飞机上鹦鹉对空姐的表现非常无礼，那个人也跟着鹦鹉学，结果两位都被丢下了飞机。这时候鹦鹉拍着翅膀对那个人说：“不会飞你可牛什么呀。”

我常听人讲：年轻热血赌一把，错了也不后悔，大不了愿赌服输。要知道愿赌服输也要你有资本才能输得起，没有资本输的人是没资格讲这个话的。

你如果年轻又没有家庭负担，那很好，这正是你输得起的资本。这个社会喜欢同情弱者，年轻就是犯错误的资本，大家都能原谅。但当你已过而立之年的时候，社会对你的容忍度就开始降低了。你必须要有面对失败的准备，要有挽回局面的能力。有些路走错了，就再也无法回头了。

我不想剥夺大家的梦想，只是想提醒你们做事情有成功更有失败，如果你输不起最好老实本分一些。

## A.2 面试官篇

### A.2.1 团队需要什么人

作为一个运维团队，技术诚然是最基本的要求。但我不会把技术排在第一位，而是更看重一个人的综合素质（包括性格、价值观、处事方法、同理心、责任心、认知力、沟通力、洞察力、执行力、学习力，等等）。原因有两个：第一，综合素质与一个人的成长教育环境有关，是受长期影响的结果的累积体现。正所谓江山易改、本性难移，这方面不是短时间就可以轻易变更的。况且企业又不是学校，没有义务和成本去做这种素质教育。第二，运维工作关系着行业生产的正常运行，如果人员素质不达标，技术能力强在产生负面影响的时候只会更加糟糕。第三，有些技术差距是受工作环境所限造成的，综合素质过硬的话，这种技术差距很容易就能弥补。我讲这些并不是为了强调技术不重要，而是提醒大家不要唯技术至上。技术可以学，而综合素质从某种程度上来讲，只能影响却不能培养。因此综合素质良好的人才是可遇而不可求的，一定要珍惜。关于这一点，A.1.4 节中，我想我已经讲得够多的了。

一般高级职位更倾向于从业五年以上有经验的人。除了基本技术要求之外，还应当具备如下能力：善于倾听，沟通协调能力强；做事规划性和规范性强；沉稳冷静，分析预见力强，考虑问题周全。

如果是招纳新人，最好能有两年左右的相关经验。需要具备一定的基础知识，另外还有三点要求：执行力强；工作及学习意愿强；做事认真负责。

### A.2.2 招聘启事中的问题

和很多虚假简历一样，我也见到过很多前后描述不一致的招聘启事，给人的感觉就是缺乏诚信，无法引起求职者的兴趣。例如有些 Title 很高的职位（××专家或高级工程师等）对应聘者的要求却简单到连新人都能 Cover 的程度。有些则是要求过于苛刻，薪资待遇却不敢恭维。还有一些职位对薪资待遇的定位非常随意。例如某专家职位薪酬 30 万 ~ 100 万。30 万也就是中高级的水平，100 万在很多公司都是 VP 的待遇了。我想求职者在面对这类职位时，对其真实性和真诚性都会秉持不信任的态度。因此 HR 应当同需求负责人与猎头公司做好有效的沟通，三方应对招聘启事的内容进行仔细的核实。

### A.2.3 面试官的修养

还是之前提到过的一个观点。面试不是企业对求职者单方面的考核，而是一次彼此都要接受检验的双向选择。一场面试下来，也许淘汰的是求职者，但可能在求职者心中，面试官和公司的形象也同样被给予了差评。一个公司的品牌形象可能来源于产品和商业操作，而我认为其企业文化形象取决于另外三个出口：前台、外务（销售、项目经理、技术支持、培训师）和 HR（包括面试官）。这三者才是还原企业内在本质的真实因素。我认为首先应该树立起一个正确的面试观，才能做好或才有资格担任面试官。有很多的个人细节问题，确实值得反思。这不仅仅是个人的修养之道，更关乎公司的对外形象，其实是有必要对面试官做一些培训的。

#### 1. 态度傲慢、自以为是

有些面试官会不自觉地有一种优越感，对候选人不是非常的友好。在交谈的过程中低着头，没有目视候选人，打断别人的讲话，语气冰冷，喜欢迟到等恶劣的行为都会引发候选人的不满。

另外有些公司还喜欢进行压力测试的考核，故意刁难或刺激候选人，然后借以观察对方的反应和心理承受能力。我认为除非岗位有这种职业需求（高层管理人员或销售），否则压力测试实在是违背常理。如果换位思考的话，你是否愿意接受这类测试呢？

#### 2. 穷追猛打、令人难堪

在某一个技术问题上，如果候选人无法回答或答错了，面试官可以稍作提示，如果提示没有效果的话，就应该转入下一个问题。此时胜负已分，就该点到为止。穷追猛打不会显得

你技高一筹，反而会令候选人更加紧张和不悦，正所谓得饶人处且饶人嘛。

### 3. 闲聊式的面试

有些面试官实在是缺乏时间观念，面试没有主题，东拉西扯，想起什么问什么，完全是漫无目的的闲聊。和候选人一样，在面试之前，面试官也应当做好充足的准备功课，问问题应做到有的放矢。

### 4. 马拉松式的面试

不要把面试弄成车轮战，更不要搞什么初试复试之类的。这又不是大奖赛，让候选人反复跑多趟是不合适的。一般职位的面试最好能够一次就完成，由技术、部门领导和 HR 三轮组成就够了。这样下来，可以把时间控制在两个小时左右。时间过长，人会感到疲惫，状态也会下降。顺便透露一个小秘密：作为一般职位，80% 的面试结果其实当场就已经决定了。如果你见到 HR 并谈到了薪水，基本上就成功 90% 了。面试结束后你很快就能收到回复。相反的，如果没有见到 HR 或部门领导，那就是没戏了。除非明确是多场次的面试，否则那种被告知回去等消息的话，就是一种委婉的拒绝。

## A.2.4 做讨人喜欢的面试官

俗话说“买卖不成仁义在”。不管结果如何，我都不希望面试过程给候选人留下不愉快的回忆。一个面试官最不愿意看到的，就是候选人在网站上的各种面试吐槽。也许面试时他们是弱势一方无法表现出自己的愤怒，可是到了网上，那些负面情绪就会全部倾泻出来。他们不仅是在表达不满，甚至还产生了对这家企业形象的敌视。对于这点我希望所有的面试官都能重视起来，做讨人喜欢的面试官。

### 1. 避免令人抓狂的问题

#### 1) 手里拿着求职者的简历要求对方做自我介绍

我实在是找不到问这个问题的充分理由。是没时间看简历？还是想核实简历的真实性？我想可能是面试官自己都不知道该说什么好，为了打开尴尬的局面并找到切入点才问的吧？我是不喜欢问这个问题的，最好还是比较体贴地留给 HR 的 MM 吧。如果这个问题让每个环节的面面试官都问上一遍，我想候选人一定会疯掉的。

#### 2) 为什么离职

这一点马云先生已经总结得很精辟了，我再补充两个比较合理的理由就是家庭原因和工作环境。除此之外，我想不到更加有说服力的理由了。所以这个问题本质上没有任何意义，稍有经验的候选人都会很有针对性的做这方面的准备，最终你得到的也只是千篇一律的巧妙说辞罢了，并不一定是对方的真实想法。

#### 3) 对我们公司了解吗

我会觉得这样的开场白真的很尴尬。即便是公司员工也不见得就了解公司的全部。有些企业文化会讲，对公司都不了解就不该来面试。但是面试官是不是因为对候选人都了解就



放弃招聘呢？我不了解，但很希望面试官能直白地告诉我这是一家怎样的企业，他们需要怎样的人，他们在做什么，他们的团队是怎样的，是否能燃起我心中的熊熊烈焰，迫不及待地想要加入这个伟大的团队。

4) 如果你被淘汰了，你觉得会是因为什么原因

我会直白地告诉面试官，那肯定是因为我与你合不来。

## 2. 什么是好的交流方式

我希望面试官能打破高高在上的那种我问你答的旧有模式。我认为让面试官先做自我介绍才是最好的开局效果。请面试官花费 5 分钟的时间先简要介绍一下公司的基本情况，还有你的团队规模、运维环境及所使用到的技术和职位要求，等等。如果是部门领导或 HR，最好能对公司的商业运作、企业文化、福利待遇做一些宣传。这种自我介绍很容易打开话题，而且还可以缓解候选人的紧张情绪，对公司或岗位有一个清晰的了解，便于接下来有针对性的交流。事实上这也是一种商业宣传，对增强候选人入职的积极性是有帮助的。退一万步讲，即便候选人没能通过考核或放弃 offer，这种待人的方式也会给对方留下一个非常美好的印象。这样做，有百利而无一害。

接下来进入技术环节，我建议问题不要超过 5 个。主要集中在三类问题上：第一，候选人在以往工作中最擅长的一两个技术，以考察候选人对以往工作是否用心钻研；第二，列出目前职位需要的技术点，让候选人选择一两个最有把握的，以考察候选人是否可以胜任工作；最后了解一下候选人最近在关注什么技术。每个问题要尽量深入，考察对方对原理或方法的理解程度。为什么不必问太多的问题呢？因为候选人挑选的已经是他最擅长的内容了，对最擅长的方面进行最深入的讨论足以考察对方的技术能力。一个问题的时间注意控制在 5 分钟以内。另外只要发现对方深入不下去了，结果也就显现出来了，应该停止讨论并切换到下一个问题。所谓点到为止，没有必要继续纠缠不清。这样计算大约总计花费半个小时。

最后再补充两个非技术问题：第一，自己最大的非技术优势是什么（举例说明）？借以考察对方的性格品质和综合能力。第二，你对新团队的期许是什么？这显然比问为什么离职要好得多。虽然我觉得这两者其实本质上并没有什么差别，但显然这样的提问更加有亲和力。

### A.2.5 团队怎样增加吸引力和凝聚力

关于团队的建设和管理方面，本人也是个初学者。因此不再过多妄言，简单地提几点建议仅供参考。

- 团队成员之间可以有差异和分层，但成员之间的技术能力不要过度分化。
- 团队可以有带头人、但不要产生绝对核心，以免出现单点隐患。
- 团队要建设导师制度，形成技术主动分享、愿意带领新人的好风气，确保成员有成长的空间。



- 设置有效的 PKI，为第二象限事务、导师制度、数据产出（降低成本或提升性能）设定 PKI。
- PKI 要具备确保有效性的三要素，包括验收标准、时间进度、完成后的奖励内容。
- 不定期进行团队内部交流，明确目标、分析问题、消除误解，去除不利于团队发展的因素。

### A.3 小结

讲了这么多，我觉得还是有很多话没有说完。这篇文章并非教科书，也并非要刻意针对什么，以上这些感悟不过是我近十年来的亲身经历和所想所得而已。我只是站在一个相对公正客观的立场之上，说了一些真诚的大实话，既讲述到了职业发展，求职攻略及关于面试官的修养问题，也希望大家就此能得到激励并重新正确地认识自己，不要昏昏沉沉地过日子，误了自己这一生。我相信我身边仍然有许多还未意识到自己的问题或希望求得帮助的朋友，这也正是我愿意鼓起勇气，奋笔疾书的最大动力。这篇文章中提到的最重要的优秀品质——用心做事的匠人态度和付诸行动的坚韧毅力，正是我人生的座右铭，这两点也希望每个人都能去践行。

最后和大家共勉一句话——也许你现在和我一样，不过是个无名小卒。但只要你愿意改变，世界一定会为你动容。

## 肖 力

《运维前线》总策划，中联润通运维总监，曾在盛大游戏和金山西山居负责系统运维工作。6年KVM虚拟化运维经验，10年游戏行业运维经验，15年运维工作经验。云技术社区创始人，《深度实践KVM》作者，《Ceph手册》译者。

## 作者介绍

**王津银** “精益运维”创始人，近10年运维经验，曾在腾讯、YY、UC从事运维工作。

**陈立军** 西山居DevOps，原新浪研发系统开发工程师。

**刘 宇** 西山居架构师，自动化运维专家，《Puppet实战》作者。

**吴传玉** 资深运维专家和虚拟化专家，10年以上x86服务器平台系统管理经验。

**余洪春** 资深运维架构师和系统管理员，10余年运维经验，《构建高可用Linux服务器》《Linux集群和自动化运维》作者。

**胥 峰** 盛大游戏高级研究员，10年运维经验，《Linux运维最佳实践》作者。

**尹会生** 西山居高级系统工程师，擅长内核调优，以及高性能和高可用性集群技术。

**张观石** YY互娱事业部运维负责人，10余年PHP开发和网站运维经验。

**彭华盛** 资深运维专家，广发银行数据中心渠道交易应用系统团队负责人。

**蒋 迪** 资深虚拟化基础架构工程师，运维专家和云计算专家。

**赵 昱** 京东金融（网银在线）支付产品技术部高级系统工程师。

**老男孩** 老男孩IT教育创始人，资深Linux技术专家。

**马 亮** 原搜狐畅游端游研发，端游、手游、运维开发主管，现腾讯云游戏资深架构师、游戏云高级产品经理。

**赵海军** 猎豹移动数据库负责人，擅长IDC运维、应用运维和数据库运维。

作为一名做了近15年技术，对运维情有独钟的“手艺人”，我深知一本好书能给人带来的益处。书中每个话题始于运维，终于运维，让人获益匪浅。我们热爱运维，希望更多运维人能从本书中受益。

——陈桂新 盛大游戏G云总负责人

这是一本关于运维人的书，读者不再只是在书中寻找答案，而是与专家们一起去经历和感受运维过程中的点滴。这是一本关于运维实践的书，它不是教程式的循规蹈矩，而是专家们将压箱底的私家干货汇总成册，诚意满满。

——余何 顺丰科技数据中心总监 《PaaS实现与运维管理》作者

由运维专家肖力领衔打造的这本书，通过14位一线专家的实践干货，从方法论、能力模型、具体实现、深度实践、团队管理等多个角度，全方位地对运维进行了阐述。相信本书能够给运维人员带来很好的启发。

——胡罡 某世界500强金融集团信息技术中心应用运行 副总经理  
Devops Master国内首批认证

作为运维人，需要不断学习、拥抱变化。运维岗以技术面广为著，“云技术社区”召集了14位一线运维专家共创佳作。本书内容广且不失深度，为广大运维人寻福谋祉。

——胡凯 bilibili运维负责人

这是一本运维界难得的好书，凝聚了国内14位运维大牛的精华，内容涉及大量丰富的实践案例，涵盖从基础设施到应用服务的各个方面，从运维的视角对每个关键路径进行详细解密，覆盖运维领域的各个分支，相信可以让不同水平层次的读者从中受益。

——刘天斯 腾讯资深运维专家/《Python自动化运维》《循序渐进学Docker》作者

不务虚，不注水，页页干货，篇篇精华。每一章都由各领域的专家撰写，作为同行，我在阅读本书的过程中，画面感很强烈，因为这些工作我也有实际经历过。SA的经历、业务运维的经历，CMDB的规划、运维部门的中长期规划，你想要的，这里都有，值得推荐！

——秦洁 安居客运维负责人

说起来有点遗憾，力哥曾邀请我一起参与本书的写作，当时忙于知数堂的工作未能参加。如今这本集合了14位奋斗在运维前线的技术专家的实践经验的好书即将面世，非常期待它能为运维圈带来一股学习和分享的新浪潮。

——叶金荣 知数堂培训联合创始人/ORACLE&MySQL ACE

